Package Summary	
javax.speech	
javax.speech.recognition	
javax.speech.synthesis	

Package javax.speech

Interface Summary	
AudioListener	The listener interface for receiving events associated with the audio input or output of an `Engine`.
AudioManager	The `AudioManager` is provided by a speech `Engine` - a `Recognizer` or `Synthesizer` - to allow an application to control audio input/output and to monitor audio-related events.
EngineCentral	Provides a list of `EngineModeDesc` objects that define the available operating modes of a speech engine.
EngineCreate	The `EngineCreate` interface is implemented by `EngineModeDesc` objects obtained through calls to the `EngineCentral` objects of each speech engine registered with the `Central` class.
Engine	The `Engine` interface is the parent interface for all speech engines including `Recognizer` and `Synthesizer`.
EngineListener	Interface defining methods to be called when state-change events for a speech engine occur.
EngineProperties	An `EngineProperties` object defines the set of run-time properties of an `Engine`.
VocabManager	Interface for management of words used by a speech `Engine`.

Class Summary

AudioAdapter	A trivial implementation of the `AudioListener` interface that receives an engine's audio events.
AudioEvent	Describes events associated with audio input/output for an `Engine`.
Central	The `Central` class is the initial access point to all speech input and output capabilities.
EngineAdapter	Trivial implementation of the `EngineListener` interface that receives a `EngineEvents`.
EngineErrorEvent	`EngineErrorEvent` is an asynchronous notification of an internal error in the engine which prevents normal behavior of that engine.
EngineEvent	`EngineEvent` notifies changes in state of a speech synthesis or recognition engine.
EngineList	`EngineList` is a container for a set of `EngineModeDesc` objects.
EngineModeDesc	`EngineModeDesc` provides information about a *specific operating mode* of a speech engine.
SpeechEvent	The root event class for all speech events.
SpeechPermission	This class represents speech permissions.
Word	The `Word` class provides a standard representation of speakable words for speech engines.

Exception Summary

AudioException	Problem encountered connecting audio to/from a speech engine.
EngineException	Signals that an error occurred while trying to create or access a speech synthesis engine, speech recognition engine or `EngineCentral` object.
SpeechException	Signals that a Java Speech API exception has occurred.
VendorDataException	Signals that a problem has been encountered loading or saving some type of vendor-specific data.

Error Summary

EngineStateError	Signals an error caused by an illegal call to a method of a speech engine.
SpeechError	Signals that an error has occurred in the javax.speech package.

Package javax.speech.recognition

Interface Summary

DictationGrammar	Provides access to the dictation capabilities of a `Recognizer`.
FinalDictationResult	Provides information on a finalized result for an utterance that matches a `DictationGrammar`.
FinalResult	`FinalResult` is an extension to the `Result` interface that provides information about a result that has been finalized - that is, recognition is complete.
FinalRuleResult	Provides information on a finalized result for an utterance that matches a `RuleGrammar`.
Grammar	Parent interface supported by all recognition grammars including `DictationGrammar` and `RuleGrammar`.
GrammarListener	A `GrammarListener` receives notifications of status change events for a `Grammar`.
RecognizerAudioListener	Extends the set of audio event of an engine for a recognizer by adding a audio level event.
Recognizer	A `Recognizer` provides access to speech recognition capabilities.
RecognizerListener	Defines an extension to the `EngineListener` interface for specific events associated with a `Recognizer`.
RecognizerProperties	Enables control of the properties of a `Recognizer`.
Result	A `Result` is issued by a `Recognizer` as it recognizes an incoming utterance that matches an active `Grammar`.
ResultListener	The methods of a `ResultListener` receive notifications of events related to a `Result` object.
ResultToken	A token (usually a word) contained by a `Result` representing something heard by a recognizer.
RuleGrammar	`RuleGrammar` interface describes a `Grammar` that defines what users may say by a set rules.
SpeakerManager	Provides control of `SpeakerProfiles` for a `Recognizer`.

Class Summary	
GrammarAdapter	The adapter which receives grammar events.
GrammarEvent	A `GrammarEvent` is issued to each `GrammarListener` attached to a `Grammar` when major events associated with that `Grammar` occur.
GrammarSyntaxDetail	Description of a problem found in a grammar usually bundled with a `GrammarException`.
RecognizerAdapter	The adapter which receives events for a `Recognizer`.
RecognizerAudioAdapter	Adaptor for a audio events of a recognizer.
RecognizerAudioEvent	Event issued to indicate detection of speech in the incoming audio stream or to periodically indicate the audio input level.
RecognizerEvent	Event issued by `Recognizer` through `RecognizerListener`.
RecognizerModeDesc	`RecognizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech recognizers.
ResultAdapter	The adapter which receives events for a `Result` object.
ResultEvent	A `ResultEvent` is issued by a `Result` object to indicate changes in the recognized tokens and changes in state.
RuleAlternatives	`RuleAlternatives` represents a `Rule` composed of a set of alternative sub-rules.
RuleCount	Attaches a count to a contained `Rule` object to indicate the number of times it may occur.
Rule	A `Rule` object is the basic component of a `RuleGrammar` and represents anything that may appear on the right-hand side of a rule definition in Java Speech Grammar Format.
RuleName	A `RuleName` is a reference to a named rule.
RuleParse	Represents the output of a parse of a `Result` or a string against a `RuleGrammar`.
RuleSequence	`RuleSequence` is a `Rule` composed of a sequence of sub-rules that must each be spoken in order.
RuleTag	`RuleTag` attaches a tag to a contained `Rule` object.
RuleToken	`RuleToken` represents speakable text in a `RuleGrammar`.
SpeakerProfile	A `SpeakerProfile` object is used to identify each enrollment by a user to a `Recognizer`.

Exception Summary	
GrammarException	Thrown if a problem is found with a Java Speech Grammar Format (JSGF) file or with a `RuleGrammar` object derived from JSGF. Grammar problems are typically identified and fixed during application development.

Error Summary	
ResultStateError	Signals an error caused by an illegal call to a method of `FinalResult`, `FinalRuleResult` or `FinalDictationResult`.

Package javax.speech.synthesis

Interface Summary

Speakable	An object implementing the `Speakable` interface can be provided to the `speak` method of a `Synthesizer` to be spoken.
SpeakableListener	The listener interface for receiving notification of events during spoken output of a `Speakable`.
Synthesizer	The `Synthesizer` interface provides primary access to speech synthesis capabilities.
SynthesizerListener	An extension to the `EngineListener` interface for receiving notification of events associated with a `Synthesizer`.
SynthesizerProperties	Provides control of the run-time properties of a `Synthesizer`.

Class Summary

SpeakableAdapter	Adapter that receives events associated with spoken output of a `Speakable` object.
SpeakableEvent	Event issued during spoken output of text.
SynthesizerAdapter	Adapter that receives events associated with a `Synthesizer`.
SynthesizerEvent	Event issued by `Synthesizer` to indicate a change in state or other activity.
SynthesizerModeDesc	`SynthesizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech synthesizers.
SynthesizerQueueItem	Represents an object on the speech output queue of a `Synthesizer`.
Voice	A description of one output voice of a speech synthesizer.

Exception Summary

JSMLException	Thrown if a syntax problem is found with text in the marked up with the Java Speech Markup Language.

Hierarchy For All Packages

Package Hierarchies:
javax.speech, javax.speech.recognition, javax.speech.synthesis

Class Hierarchy

- class java.lang.Object
  - class java.util.AbstractCollection(implements java.util.Collection)
    - class java.util.AbstractList(implements java.util.List)
      - class java.util.Vector(implements java.lang.Cloneable, java.util.List, java.io.Serializable)
        - class javax.speech.**EngineList**
  - class javax.speech.**AudioAdapter**(implements javax.speech.AudioListener)
    - class javax.speech.recognition.**RecognizerAudioAdapter**(implements javax.speech.recognition.RecognizerAudioListener)
  - class javax.speech.**Central**
  - class javax.speech.**EngineAdapter**(implements javax.speech.EngineListener)
    - class javax.speech.recognition.**RecognizerAdapter**(implements javax.speech.recognition.RecognizerListener)
    - class javax.speech.synthesis.**SynthesizerAdapter**(implements javax.speech.synthesis.SynthesizerListener)
  - class javax.speech.**EngineModeDesc**
    - class javax.speech.recognition.**RecognizerModeDesc**
    - class javax.speech.synthesis.**SynthesizerModeDesc**
  - class java.util.EventObject(implements java.io.Serializable)
    - class javax.speech.**SpeechEvent**
      - class javax.speech.**AudioEvent**
        - class javax.speech.recognition.**RecognizerAudioEvent**
      - class javax.speech.**EngineEvent**
        - class javax.speech.**EngineErrorEvent**
        - class javax.speech.recognition.**RecognizerEvent**
        - class javax.speech.synthesis.**SynthesizerEvent**
      - class javax.speech.recognition.**GrammarEvent**
      - class javax.speech.recognition.**ResultEvent**
      - class javax.speech.synthesis.**SpeakableEvent**
  - class javax.speech.recognition.**GrammarAdapter**(implements javax.speech.recognition.GrammarListener)
  - class javax.speech.recognition.**GrammarSyntaxDetail**
  - class java.security.Permission(implements java.security.Guard, java.io.Serializable)
    - class java.security.BasicPermission(implements java.io.Serializable)
      - class javax.speech.**SpeechPermission**
  - class javax.speech.recognition.**ResultAdapter**(implements javax.speech.recognition.ResultListener)
  - class javax.speech.recognition.**Rule**(implements java.io.Serializable)
    - class javax.speech.recognition.**RuleAlternatives**

  - class javax.speech.recognition.**RuleCount**
  - class javax.speech.recognition.**RuleName**
  - class javax.speech.recognition.**RuleParse**
  - class javax.speech.recognition.**RuleSequence**
  - class javax.speech.recognition.**RuleTag**
  - class javax.speech.recognition.**RuleToken**
- class javax.speech.synthesis.**SpeakableAdapter**(implements javax.speech.synthesis.SpeakableListener)
- class javax.speech.recognition.**SpeakerProfile**
- class javax.speech.synthesis.**SynthesizerQueueItem**
- class java.lang.Throwable(implements java.io.Serializable)
  - class java.lang.Error
    - class javax.speech.**SpeechError**
      - class javax.speech.**EngineStateError**
      - class javax.speech.recognition.**ResultStateError**
  - class java.lang.Exception
    - class javax.speech.**SpeechException**
      - class javax.speech.**AudioException**
      - class javax.speech.**EngineException**
      - class javax.speech.recognition.**GrammarException**
      - class javax.speech.synthesis.**JSMLException**
      - class javax.speech.**VendorDataException**
- class javax.speech.synthesis.**Voice**(implements java.lang.Cloneable)
- class javax.speech.**Word**

Interface Hierarchy

- interface javax.speech.**AudioManager**
- interface javax.speech.**EngineCentral**
- interface javax.speech.**EngineCreate**
- interface javax.speech.**Engine**
  - interface javax.speech.recognition.**Recognizer**
  - interface javax.speech.synthesis.**Synthesizer**
- interface javax.speech.**EngineProperties**
  - interface javax.speech.recognition.**RecognizerProperties**
  - interface javax.speech.synthesis.**SynthesizerProperties**
- interface java.util.EventListener
  - interface javax.speech.**AudioListener**
    - interface javax.speech.recognition.**RecognizerAudioListener**
  - interface javax.speech.**EngineListener**
    - interface javax.speech.recognition.**RecognizerListener**
    - interface javax.speech.synthesis.**SynthesizerListener**
  - interface javax.speech.recognition.**GrammarListener**
  - interface javax.speech.recognition.**ResultListener**
  - interface javax.speech.synthesis.**SpeakableListener**
- interface javax.speech.recognition.**Grammar**
  - interface javax.speech.recognition.**DictationGrammar**

  - interface javax.speech.recognition.**RuleGrammar**
- interface javax.speech.recognition.**Result**
  - interface javax.speech.recognition.**FinalResult**
    - interface javax.speech.recognition.**FinalDictationResult**
    - interface javax.speech.recognition.**FinalRuleResult**
- interface javax.speech.recognition.**ResultToken**
- interface javax.speech.synthesis.**Speakable**
- interface javax.speech.recognition.**SpeakerManager**
- interface javax.speech.**VocabManager**

Package javax.speech

Interface Summary	
AudioListener	The listener interface for receiving events associated with the audio input or output of an `Engine`.
AudioManager	The `AudioManager` is provided by a speech `Engine` - a `Recognizer` or `Synthesizer` - to allow an application to control audio input/output and to monitor audio-related events.
EngineCentral	Provides a list of `EngineModeDesc` objects that define the available operating modes of a speech engine.
EngineCreate	The `EngineCreate` interface is implemented by `EngineModeDesc` objects obtained through calls to the `EngineCentral` objects of each speech engine registered with the `Central` class.
Engine	The `Engine` interface is the parent interface for all speech engines including `Recognizer` and `Synthesizer`.
EngineListener	Interface defining methods to be called when state-change events for a speech engine occur.
EngineProperties	An `EngineProperties` object defines the set of run-time properties of an `Engine`.
VocabManager	Interface for management of words used by a speech `Engine`.

Class Summary	
AudioAdapter	A trivial implementation of the `AudioListener` interface that receives an engine's audio events.
AudioEvent	Describes events associated with audio input/output for an `Engine`.
Central	The `Central` class is the initial access point to all speech input and output capabilities.
EngineAdapter	Trivial implementation of the `EngineListener` interface that receives a `EngineEvents`.
EngineErrorEvent	`EngineErrorEvent` is an asynchronous notification of an internal error in the engine which prevents normal behavior of that engine.
EngineEvent	`EngineEvent` notifies changes in state of a speech synthesis or recognition engine.
EngineList	`EngineList` is a container for a set of `EngineModeDesc` objects.
EngineModeDesc	`EngineModeDesc` provides information about a *specific operating mode* of a speech engine.
SpeechEvent	The root event class for all speech events.
SpeechPermission	This class represents speech permissions.
Word	The `Word` class provides a standard representation of speakable words for speech engines.

Exception Summary	
AudioException	Problem encountered connecting audio to/from a speech engine.
EngineException	Signals that an error occurred while trying to create or access a speech synthesis engine, speech recognition engine or `EngineCentral` object.
SpeechException	Signals that a Java Speech API exception has occurred.
VendorDataException	Signals that a problem has been encountered loading or saving some type of vendor-specific data.

Error Summary	
EngineStateError	Signals an error caused by an illegal call to a method of a speech engine.
SpeechError	Signals that an error has occurred in the javax.speech package.

Class javax.speech.AudioAdapter

```
java.lang.Object
  |
  +--javax.speech.AudioAdapter
```

Subclasses:
RecognizerAudioAdapter

---

public class **AudioAdapter**
extends java.lang.Object
implements AudioListener

A trivial implementation of the `AudioListener` interface that receives an engine's audio events. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Extended by the `RecognizerAudioAdapter` that adds specialized audio events for a `Recognizer`.

Note: until the Java Sound API is finalized, the `AudioManager` and other audio classes and interfaces will remain as placeholders for future expansion. Only the `Recognizer` audio events are functional in this release.

See Also:
RecognizerAudioAdapter

Constructor Summary

AudioAdapter()	

Methods inherited from class java.lang.Object	
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait	

Constructor Detail

AudioAdapter

```
public AudioAdapter()
```

Class javax.speech.AudioEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.AudioEvent
```

Subclasses:
RecognizerAudioEvent

---

public class **AudioEvent**
extends SpeechEvent

Describes events associated with audio input/output for an `Engine`. The event source is an `Engine` object.

Extended by the `RecognizerAudioEvent` class that provides specialized events for a `Recognizer`.

Note: until the Java Sound API is finalized, the `AudioManager` and other audio classes and interfaces will remain as placeholders for future expansion. Only the `Recognizer` audio events are functional in this release.

See Also:
Engine, RecognizerAudioEvent, Serialized Form

Fields inherited from class javax.speech.SpeechEvent	
id	

Fields inherited from class java.util.EventObject	
source	

Constructor Summary

AudioEvent(Engine source, int id) Constructs an AudioEvent with a specified id.	

Method Summary

java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

AudioEvent

```
public AudioEvent(Engine source,
                  int id)
```

Constructs an AudioEvent with a specified id.

Parameters:
`source` - `Engine` that produced the event
`id` - type of audio event

Method Detail

paramString

```
public java.lang.String paramString()
```

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event

Overrides:
paramString in class SpeechEvent

Class javax.speech.AudioException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
                    |
                    +--javax.speech.AudioException
```

---

public class **AudioException**
extends SpeechException

Problem encountered connecting audio to/from a speech engine.

See Also:
Serialized Form

Constructor Summary
AudioException() Constructs a `AudioException` with no detail message.
AudioException(java.lang.String s) Construct an `AudioException` with the specified detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

AudioException

`public AudioException()`

Constructs a `AudioException` with no detail message.

AudioException

`public AudioException(java.lang.String s)`

Construct an `AudioException` with the specified detail message. The string describes this particular exception.

Parameters:
`s` - the detail message

Interface javax.speech.AudioListener

Subinterfaces:
RecognizerAudioListener

Implementing Classes:
AudioAdapter

---

public abstract interface **AudioListener**
extends java.util.EventListener

The listener interface for receiving events associated with the audio input or output of an `Engine`. An `AudioListener` is attached to an `Engine` by the `addAudioListener` method of the engine's `AudioManager`.

`RecognizerAudioListener` extends this interface to support `RecognizerAudioEvents` provided by a `Recognizer`.

Note: until the Java Sound API is finalized, the `AudioManager` and other audio classes and interfaces will remain as placeholders for future expansion. Only the `Recognizer` audio events are functional in this release.

See Also:
addAudioListener, RecognizerAudioEvent, RecognizerAudioListener

Interface javax.speech.AudioManager

public abstract interface **AudioManager**

The `AudioManager` is provided by a speech `Engine` - a `Recognizer` or `Synthesizer` - to allow an application to control audio input/output and to monitor audio-related events. The `AudioManager` for an engine is obtained by calling its `getAudioManager` method.

Note: until the Java Sound API is finalized, the `AudioManager` and other audio classes and interfaces will remain as placeholders for future expansion. Only the `Recognizer` audio events are functional in this release.

See Also:
getAudioManager

Method Summary

void	**addAudioListener**(AudioListener listener) Request notifications of audio events to an `AudioListener`.
void	**removeAudioListener**(AudioListener listener) Detach an audio listener from this `AudioManager`.

Method Detail

addAudioListener

`public void addAudioListener(AudioListener listener)`

Request notifications of audio events to an `AudioListener`. An application can attach multiple audio listeners to an `AudioManager`. If the engine is a `Recognizer`, a `RecognizerAudioListener` may be attached since the `RecognizerAudioListener` interface extends the `AudioListener` interface.

See Also:
AudioListener, RecognizerAudioListener

removeAudioListener

`public void removeAudioListener(AudioListener listener)`

Detach an audio listener from this `AudioManager`.

Class javax.speech.Central

```
java.lang.Object
  |
  +--javax.speech.Central
```

public class **Central**
extends java.lang.Object

The `Central` class is the initial access point to all speech input and output capabilities. `Central` provides the ability to locate, select and create speech recognizers and speech synthesizers

Creating a `Recognizer` or `Synthesizer`

The `createRecognizer` and `createSynthesizer` methods are used to create speech engines. Both methods accept a single parameter that defines the required properties for the engine to be created. The parameter is an `EngineModeDesc` and may be one of the sub-classes: `RecognizerModeDesc` or `SynthesizerModeDesc`.

A mode descriptor defines a set of required properties for an engine. For example, a `SynthesizerModeDesc` can describe a `Synthesizer` for Swiss German that has a male voice. Similarly, a `RecognizerModeDesc` can describe a `Recognizer` that supports dictation for Japanese.

An application is responsible for determining its own functional requirements for speech input/output and providing an appropriate mode descriptor. There are three cases for mode descriptors:

1. `null`
2. Created by the application
3. Obtained from the `availableRecognizers` or `availableSynthesizers` methods of `Central`.

The mode descriptor is passed to the `createRecognizer` or `createSynthesizer` methods of `Central` to create a `Recognizer` or `Synthesizer`. The created engine matches all the engine properties in the mode descriptor passed to the create method. If no suitable speech engine is available, the create methods return `null`.

The create engine methods operate differently for the three cases. That is, engine selection depends upon the type of the mode descriptor:

1. `null` mode descriptor: the `Central` class selects a suitable engine for the default `Locale`.
2. Application-created mode descriptor: the `Central` class attempts to locate an engine with all application-specified properties.
3. Mode descriptor from `availableRecognizers` or `availableSynthesizers`: descriptors returned by these two methods identify a specific engine with a specific operating mode. `Central` creates an instance of that engine. (Note: these mode descriptors are distinguished because they implement the `EngineCreate` interface.)

Case 1: Example

```
// Create a synthesizer for the default Locale
Synthesizer synth = Central.createSynthesizer(null);
```

Case 2: Example

```
// Create a dictation recognizer for British English
// Note: the UK locale is English spoken in Britain
RecognizerModeDesc desc = new RecognizerModeDesc(Locale.UK, Boolean.TRUE);
Recognizer rec = Central.createRecognizer(desc);
```

Case 3: Example

```
// Obtain a list of all German recognizers
RecognizerModeDesc desc = new RecognizerModeDesc(Locale.GERMAN);
EngineList list = Central.availableRecognizers(desc);
// select amongst by other desired engine properties
RecognizerModeDesc chosen = ...
// create an engine from "chosen" - an engine-provided descriptor
Recognizer rec = Central.createRecognizer(chosen);
```

Engine Selection Procedure: Cases 1 & 2

For cases 1 and 2 there is a defined procedure for selecting an engine to be created. (For case 3, the application can apply it's own selection procedure.)

`Locale` is treated specially in the selection to ensure that language is always considered when selecting an engine. If a locale is not provided, the default locale (`java.util.Locale.getDefault`) is used.

The selection procedure is:

1. If the locale is undefined add the language of the default locale to the required properties.
2. If a `Recognizer` or `Synthesizer` has been created already and it has the required properties, return a reference to it. (The last created engine is checked.)
3. Obtain a list of all recognizer or synthesizer modes that match the required properties.
4. Amongst the matching engines, give preference to:
    - A running engine (`EngineModeDesc.getRunning` is `true`),
    - An engine that matches the default locale's country.

When more than one engine is a legal match in the final step, the engines are ordered as returned by the `availableRecognizers` or `availableSynthesizers` method.

Security

Access to speech engines is restricted by Java's security system. This is to ensure that malicious applets don't use the speech engines inappropriately. For example, a recognizer should not be usable without explicit permission because it could be used to monitor ("bug") an office.

A number of methods throughout the API throw `SecurityException`. Individual implementations of `Recognizer` and `Synthesizer` may throw `SecurityException` on additional methods as required to protect a client from malicious applications and applets.

The `SpeechPermission` class defines the types of permission that can be granted or denied for applications. This permission system is based on the JDK 1.2 fine-grained security model.

Engine Registration

The `Central` class locates, selects and creates speech engines from amongst a list of registered engines. Thus, for an engine to be used by Java applications, the engine must register itself with `Central`. There are two registration mechanisms: (1) add an `EngineCentral` class to a speech properties file, (2) temporarily register an engine by calling the `registerEngineCentral` method.

The speech properties files provide *persistent* registration of speech engines. When `Central` is first called, it looks for properties in two files:

```
<user.home>/speech.properties
<java.home>/lib/speech.properties
```

where the `<user.home>` and `<java.home>` are the values obtained from the `System` properties object. (The '/' separator will vary across operating systems.) Engines identified in either properties file are made available through the methods of `Central`.

The property files must contain data in the format that is read by the `load` method of the `Properties` class. `Central` looks for properties of the form

```
com.acme.recognizer.EngineCentral=com.acme.recognizer.AcmeEngineCentral
```

This line is interpreted as "the `EngineCentral` object for the `com.acme.recognizer` engine is the class called `com.acme.recognizer.AcmeEngineCentral`. When it is first called, the `Central` class will attempt to create an instance of each `EngineCentral` object and will ensure that it implements the `EngineCentral` interface.

Note to engine providers: `Central` calls each `EngineCentral` for each call to `availableRecognizers` or `availableSynthesizers` and sometimes `createRecognizer` and `createSynthesizer` The results are not stored. The `EngineCentral.createEngineList` method should be reasonably efficient.

Method Summary

static EngineList	**availableRecognizers**(EngineModeDesc require) List `EngineModeDesc` objects for available recognition engine modes that match the required properties.
static EngineList	**availableSynthesizers**(EngineModeDesc require) List `EngineModeDesc` objects for available synthesis engine modes that match the required properties.
static Recognizer	**createRecognizer**(EngineModeDesc require) Create a `Recognizer` with specified required properties.
static Synthesizer	**createSynthesizer**(EngineModeDesc require) Create a `Synthesizer` with specified required properties.
static void	**registerEngineCentral**(java.lang.String className) Register a speech engine with the `Central` class for use by the current application.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

createRecognizer

```
public static final Recognizer createRecognizer(EngineModeDesc require)
                                         throws java.lang.IllegalArgumentException,
                                                EngineException,
                                                java.lang.SecurityException
```

Create a `Recognizer` with specified required properties. If there is no `Recognizer` with the required properties the method returns `null`.

The required properties defined in the input parameter may be provided as either an `EngineModeDesc` object or a `RecognizerModeDesc` object. The input parameter may also be `null`, in which case an engine is selected that supports the language of the default locale.

A non-null mode descriptor may be either application-created or a mode descriptor returned by the `availableRecognizers` method.

The mechanisms for creating a `Recognizer` are described above in detail.

Parameters:
`require` - required engine properties or `null` for default engine selection
Returns:
a recognizer matching the required properties or `null` if none is available
Throws:
java.lang.IllegalArgumentException - if the properties of the `EngineModeDesc` do not refer to a known engine or engine mode.
EngineException - if the engine defined by this `RecognizerModeDesc` could not be properly created.
java.lang.SecurityException - if the caller does not have `createRecognizer` permission
See Also:
availableRecognizers, RecognizerModeDesc

---

availableRecognizers

```
public static final EngineList availableRecognizers(EngineModeDesc require)
                                             throws java.lang.SecurityException
```

List `EngineModeDesc` objects for available recognition engine modes that match the required properties. If the `require` parameter is `null`, then all known recognizers are listed.

Returns a zero-length list if no engines are available or if no engines have the required properties. (The method never returns null).

The order of the `EngineModeDesc` objects in the list is partially defined. For each registered engine (technically, each registered `EngineCentral` object) the order of the descriptors is preserved. Thus, each installed speech engine should order its descriptor objects with the most useful modes first, for example, a mode that is already loaded and running on a desktop.

Parameters:
`require` - an `EngineModeDesc` or `RecognizerModeDesc` defining the required features of the mode descriptors in the returned list
Returns:
list of mode descriptors with the required properties
Throws:
java.lang.SecurityException - if the caller does not have permission to use speech recognition

---

createSynthesizer

```
public static final Synthesizer createSynthesizer(EngineModeDesc require)
                                           throws java.lang.IllegalArgumentException,
                                                  EngineException
```

Create a `Synthesizer` with specified required properties. If there is no `Synthesizer` with the required properties the method returns `null`.

The required properties defined in the input parameter may be provided as either an `EngineModeDesc` object or a `SynthesizerModeDesc` object. The input parameter may also be `null`, in which case an engine is selected that supports the language of the default locale.

A non-null mode descriptor may be either application-created or a mode descriptor returned by the `availableSynthesizers` method.

The mechanisms for creating a `Synthesizer` are described above in detail.

Parameters:
`require` - required engine properties or `null` for default engine selection
Returns:
a `Synthesizer` matching the required properties or `null` if none is available
Throws:
java.lang.IllegalArgumentException - if the properties of the `EngineModeDesc` do not refer to a known engine or engine mode.
EngineException - if the engine defined by this `SynthesizerModeDesc` could not be properly created.
See Also:
availableSynthesizers, SynthesizerModeDesc

---

availableSynthesizers

```
public static final EngineList availableSynthesizers(EngineModeDesc require)
                                                     throws java.lang.SecurityException
```

List `EngineModeDesc` objects for available synthesis engine modes that match the required properties. If the `require` parameter is `null`, then all available known synthesizers are listed.

Returns an empty list (rather than null) if no engines are available or if no engines have the required properties.

The order of the `EngineModeDesc` objects in the list is partially defined. For each speech installation (technically, each registered `EngineCentral` object) the order of the descriptors is preserved. Thus, each installed speech engine should order its descriptor objects with the most useful modes first, for example, a mode that is already loaded and running on a desktop.

Throws:

java.lang.SecurityException - if the caller does not have permission to use speech engines

---

registerEngineCentral

```
public static final void registerEngineCentral(java.lang.String className)
                                       throws EngineException
```

Register a speech engine with the `Central` class for use by the current application. This call adds the specified class name to the list of `EngineCentral` objects. The registered engine is **not** stored persistently in the properties files. If `className` is already registered, the call has no effect.

The class identified by `className` must have an empty constructor.

Parameters:

`className` - name of a class that implements the `EngineCentral` interface and provides access to an engine implementation

Throws:

EngineException - if `className` is not a legal class or it does not implement the `EngineCentral` interface

---

Interface javax.speech.Engine

Subinterfaces:

Recognizer, Synthesizer

public abstract interface **Engine**

The `Engine` interface is the parent interface for all speech engines including `Recognizer` and `Synthesizer`. A speech engine is a generic entity that either processes speech input or produces speech output. Engines - recognizers and synthesizers - derive the following functionality from the `Engine` interface:

- `allocate` and `deallocate` methods.
- `pause` and `resume` methods.
- Access to a `AudioManager` and `VocabManager`.
- Access to `EngineProperties`.
- Access to the engine's `EngineModeDesc`.
- Methods to add and remove `EngineListener` objects.

Engines are located, selected and created through methods of the `Central` class.

Engine State System: Allocation

Each type of speech engine has a well-defined set of states of operation, and well-defined behavior for moving between states. These states are defined by constants of the `Engine`, `Recognizer` and `Synthesizer` interfaces.

The `Engine` interface defines three methods for viewing and monitoring states: `getEngineState`, `waitEngineState` and `testEngineState`. An `EngineEvent` is issued to `EngineListeners` each time an `Engine` changes state.

The basic states of any speech engine (`Recognizer` or `Synthesizer`) are `DEALLOCATED`, `ALLOCATED`, `ALLOCATING_RESOURCES` and `DEALLOCATING_RESOURCES`. An engine in the `ALLOCATED` state has acquired all the resources it requires to perform its core functions.

Engines are created in the `DEALLOCATED` state and a call to `allocate` is required to prepare them for usage. The `ALLOCATING_RESOURCES` state is an intermediate state between `DEALLOCATED` and `ALLOCATED` which an engine occupies during the resource allocation process (which may be a very short period or takes 10s of seconds).

Once an application finishes using a speech engine it should always explicitly free system resources by calling the `deallocate` method. This call transitions the engine to the `DEALLOCATED` state via some period in the `DEALLOCATING_RESOURCES` state.

The methods of `Engine`, `Recognizer` and `Synthesizer` perform differently according to the engine's allocation state. Many methods cannot be performed when an engine is in either the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state. Many methods block (wait) for an engine in the `ALLOCATING_RESOURCES` state until the engine reaches the `ALLOCATED` state. This blocking/exception behavior is defined separately for each method of `Engine`, `Synthesizer` and `Recognizer`.

Engine State System: Sub-states of `ALLOCATED`

The `ALLOCATED` states has sub-states. (The `DEALLOCATED`, `ALLOCATING_RESOURCES` and `DEALLOCATING_RESOURCES` states do not have any sub-states.)

- Any `ALLOCATED` engine (`Recognizer` or `Synthesizer`) is either `PAUSED` or `RESUMED`. These state indicates whether audio input/output is stopped or running.
- An `ALLOCATED` `Synthesizer` has additional sub-states for `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` that indicate the status of its speech output queue. These two states are independent of the `PAUSED` and `RESUMED` states.
- An `ALLOCATED` `Recognizer` has additional sub-states for `LISTENING`, `PROCESSING` and `SUSPENDED` that indicate the status of the recognition process. These three states are independent of the `PAUSED` and `RESUMED` states (with the exception of minor interactions documented with `Recognizer`).
- An `ALLOCATED` `Recognizer` also has additional sub-states for `FOCUS_ON` and `FOCUS_OFF`. Focus determines when most of an application's grammars are active or deactive for recognition. The focus states are independent of the `PAUSED` and `RESUMED` states and of the `LISTENING/PROCESSING/SUSPENDED` states. (Limited exceptions are discussed in the documentation for `Recognizer`).

The `pause` and `resume` methods are used to transition an engine between the `PAUSED` and `RESUMED` states. The `PAUSED` and `RESUMED` states are shared by all applications that use the underlying engine. For instance, pausing a recognizer pauses all applications that use that engine.

Engine State System: get/test/wait

The current state of an `Engine` is returned by the `getEngineState` method. The `waitEngineState` method blocks the calling thread until the `Engine` reaches a specified state. The `testEngineState` tests whether an `Engine` is in a specified state.

The state values can be bitwise OR'ed (using the Java "|" operator). For example, for an allocated, resumed synthesizer with items in its speech output queue, the state is

```
Engine.ALLOCATED | Engine.RESUMED | Synthesizer.QUEUE_NOT_EMPTY
```

The states and sub-states defined above put constraints upon the state of an engine. The following are examples of illegal states:

Illegal `Engine` states:
    `Engine.DEALLOCATED | Engine.RESUMED`
    `Engine.ALLOCATED | Engine.DEALLOCATED`
Illegal `Synthesizer` states:
    `Engine.DEALLOCATED | Engine.QUEUE_NOT_EMPTY`
    `Engine.QUEUE_EMPTY | Engine.QUEUE_NOT_EMPTY`
Illegal `Recognizer` states:
    `Engine.DEALLOCATED | Engine.PROCESSING`
    `Engine.PROCESSING | Engine.SUSPENDED`

Calls to the `testEngineState` and `waitEngineState` methods with illegal state values cause an exception to be thrown.

See Also:
Central, Synthesizer, Recognizer

Field Summary

static long	**ALLOCATED** Bit of state that is set when an `Engine` is in the allocated state.
static long	**ALLOCATING_RESOURCES** Bit of state that is set when an `Engine` is being allocated - the transition state between `DEALLOCATED` to `ALLOCATED` following a call to the `allocate` method.
static long	**DEALLOCATED** Bit of state that is set when an `Engine` is in the deallocated state.
static long	**DEALLOCATING_RESOURCES** Bit of state that is set when an `Engine` is being deallocated - the transition state between `ALLOCATED` to `DEALLOCATED`.
static long	**PAUSED** Bit of state that is set when an `Engine` is is in the `ALLOCATED` state and is `PAUSED`.
static long	**RESUMED** Bit of state that is set when an `Engine` is is in the `ALLOCATED` state and is `RESUMED`.

Method Summary

void	**addEngineListener**(EngineListener listener) Request notifications of events of related to the `Engine`.
void	**allocate**() Allocate the resources required for the `Engine` and put it into the `ALLOCATED` state.
void	**deallocate**() Free the resources of the engine that were acquired during allocation and during operation and return the engine to the `DEALLOCATED`.
AudioManager	**getAudioManager**() Return an object which provides management of the audio input or output for the `Engine`.
EngineModeDesc	**getEngineModeDesc**() Return a mode descriptor that defines the operating properties of the engine.
EngineProperties	**getEngineProperties**() Return the `EngineProperties` object (a JavaBean).
long	**getEngineState**() Returns a or'ed set of flags indicating the current state of an `Engine`.
VocabManager	**getVocabManager**() Return an object which provides management of the vocabulary for the `Engine`.
void	**pause**() Pause the audio stream for the engine and put the `Engine` into the `PAUSED` state.
void	**removeEngineListener**(EngineListener listener) Remove a listener from this `Engine`.
void	**resume**() Put the `Engine` in the `RESUMED` state to resume audio streaming to or from a paused engine.
boolean	**testEngineState**(long state) Returns true if the current engine state matches the specified state.
void	**waitEngineState**(long state) Blocks the calling thread until the `Engine` is in a specified state.

Field Detail

DEALLOCATED

`public static final long` **`DEALLOCATED`**

Bit of state that is set when an `Engine` is in the deallocated state. A deallocated engine does not have the resources necessary for it to carry out its basic functions.

In the `DEALLOCATED` state, many of the methods of an `Engine` throw an exception when called. The `DEALLOCATED` state has no sub-states.

An `Engine` is always created in the `DEALLOCATED` state. A `DEALLOCATED` can transition to the `ALLOCATED` state via the `ALLOCATING_RESOURCES` state following a call to the `allocate` method. An `Engine` returns to the `DEALLOCATED` state via the `DEALLOCATING_RESOURCES` state with a call to the `deallocate` method.

See Also:
allocate, deallocate, getEngineState, waitEngineState

ALLOCATING_RESOURCES

`public static final long` **`ALLOCATING_RESOURCES`**

Bit of state that is set when an `Engine` is being allocated - the transition state between `DEALLOCATED` to `ALLOCATED` following a call to the `allocate` method. The `ALLOCATING_RESOURCES` state has no sub-states. In the `ALLOCATING_RESOURCES` state, many of the methods of `Engine`, `Recognizer`, and `Synthesizer` will block until the `Engine` reaches the `ALLOCATED` state and the action can be performed.

See Also:
getEngineState, waitEngineState

ALLOCATED

`public static final long` **`ALLOCATED`**

Bit of state that is set when an `Engine` is in the allocated state. An engine in the `ALLOCATED` state has acquired the resources required for it to carry out its core functions.

The `ALLOCATED` states has sub-states for `RESUMED` and `PAUSED`. Both `Synthesizer` and `Recognizer` define additional sub-states of `ALLOCATED`.

An `Engine` is always created in the `DEALLOCATED` state. It reaches the `ALLOCATED` state via the `ALLOCATING_RESOURCES` state with a call to the `allocate` method.

See Also:
Synthesizer, Recognizer, getEngineState, waitEngineState

DEALLOCATING_RESOURCES

```
public static final long DEALLOCATING_RESOURCES
```

Bit of state that is set when an `Engine` is being deallocated - the transition state between `ALLOCATED` to `DEALLOCATED`. The `DEALLOCATING_RESOURCES` state has no sub-states. In the `DEALLOCATING_RESOURCES` state, most methods of `Engine`, `Recognizer` and `Synthesizer` throw an exception.

See Also:
getEngineState, waitEngineState

PAUSED

```
public static final long PAUSED
```

Bit of state that is set when an `Engine` is is in the `ALLOCATED` state and is `PAUSED`. In the `PAUSED` state, audio input or output stopped.

An `ALLOCATED` engine is always in either in the `PAUSED` or `RESUMED`. The `PAUSED` and `RESUMED` states are sub-states of the `ALLOCATED` state.

See Also:
RESUMED, ALLOCATED, getEngineState, waitEngineState

RESUMED

```
public static final long RESUMED
```

Bit of state that is set when an `Engine` is is in the `ALLOCATED` state and is `RESUMED`. In the `RESUMED` state, audio input or output active.

An `ALLOCATED` engine is always in either in the `PAUSED` or `RESUMED`. The `PAUSED` and `RESUMED` states are sub-states of the `ALLOCATED` state.

See Also:
RESUMED, ALLOCATED, getEngineState, waitEngineState

Method Detail

getEngineState

```
public long getEngineState()
```

Returns a or’ed set of flags indicating the current state of an `Engine`. The format of the returned `state` value is described above.

An `EngineEvent` is issued each time the `Engine` changes state.

The `getEngineState` method can be called successfully in any `Engine` state.

See Also:
getEngineState, waitEngineState, getNewEngineState, getOldEngineState

---

waitEngineState

```
public void waitEngineState(long state)
                     throws java.lang.InterruptedException,
                            java.lang.IllegalArgumentException
```

Blocks the calling thread until the `Engine` is in a specified state. The format of the `state` value is described above.

All state bits specified in the `state` parameter must be set in order for the method to return, as defined for the `testEngineState` method. If the `state` parameter defines an unreachable state (e.g. `PAUSED | RESUMED`) an exception is thrown.

The `waitEngineState` method can be called successfully in any `Engine` state.

Throws:
java.lang.InterruptedException - if another thread has interrupted this thread.
java.lang.IllegalArgumentException - if the specified state is unreachable
See Also:
testEngineState, getEngineState

---

testEngineState

```
public boolean testEngineState(long state)
                        throws java.lang.IllegalArgumentException
```

Returns true if the current engine state matches the specified state. The format of the `state` value is described above.

The test performed is not an exact match to the current state. Only the specified states are tested. For example the following returns true only if the `Synthesizer` queue is empty, irrespective of the pause/resume and allocation states.

```
if (synth.testEngineState(Synthesizer.QUEUE_EMPTY)) ...
```

The `testEngineState` method is equivalent to:

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

The `testEngineState` method can be called successfully in any `Engine` state.

Throws:
java.lang.IllegalArgumentException - if the specified state is unreachable

---

allocate

```
public void allocate()
              throws EngineException,
                     EngineStateError
```

Allocate the resources required for the `Engine` and put it into the `ALLOCATED` state. When this method returns successfully the `ALLOCATED` bit of engine state is set, and the `testEngineState(Engine.ALLOCATED)` method returns `true`. During the processing of the method, the `Engine` is temporarily in the `ALLOCATING_RESOURCES` state.

When the `Engine` reaches the `ALLOCATED` state other engine states are determined:

- `PAUSED` or `RESUMED`: the pause state depends upon the existing state of the engine. In a multi-app environment, the pause/resume state of the engine is shared by all apps.
- A `Recognizer` always starts in the `LISTENING` state when newly allocated but may transition immediately to another state.
- A `Recognizer` may be allocated in either the `HAS_FOCUS` state or `LOST_FOCUS` state depending upon the activity of other applications.
- A `Synthesizer` always starts in the `QUEUE_EMPTY` state when newly allocated.

While this method is being processed events are issued to any `EngineListeners` attached to the `Engine` to indicate state changes. First, as the `Engine` changes from the `DEALLOCATED` to the `ALLOCATING_RESOURCES` state, an `ENGINE_ALLOCATING_RESOURCES` event is issued. As the allocation process completes, the engine moves from the `ALLOCATING_RESOURCES` state to the `ALLOCATED` state and an `ENGINE_ALLOCATED` event is issued.

The `allocate` method should be called for an `Engine` in the `DEALLOCATED` state. The method has no effect for an `Engine` is either the `ALLOCATING_RESOURCES` or `ALLOCATED` states. The method throws an exception in the `DEALLOCATING_RESOURCES` state.

If any problems are encountered during the allocation process so that the engine cannot be allocated, the engine returns to the `DEALLOCATED` state (with an `ENGINE_DEALLOCATED` event), and an `EngineException` is thrown.

Allocating the resources for an engine may be fast (less than a second) or slow (several 10s of seconds) depending upon a range of factors. Since the `allocate` method does not return until allocation is completed applications may want to perform allocation in a background thread and proceed with other activities. The following code uses an inner class implementation of `Runnable` to create a background thread for engine allocation:

```
static Engine engine;

public static void main(String argv[])
{
  engine = Central.createRecognizer();

  new Thread(new Runnable() {
      public void run() {
        engine.allocate();
      }
    }).start();

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

```
  ...

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

Throws:
EngineException - if an allocation error occurred or the engine is not operational.
EngineStateError - if called for an engine in the `DEALLOCATING_RESOURCES` state

See Also:
getEngineState, deallocate, ALLOCATED, ENGINE_ALLOCATED

---

deallocate

```
public void deallocate()
              throws EngineException,
                     EngineStateError
```

Free the resources of the engine that were acquired during allocation and during operation and return the engine to the `DEALLOCATED`. When this method returns the `DEALLOCATED` bit of engine state is set so the `testEngineState(Engine.DEALLOCATED)` method returns `true`. During the processing of the method, the `Engine` is temporarily in the `DEALLOCATING_RESOURCES` state.

A deallocated engine can be re-started with a subsequent call to `allocate`.

Engines need to clean up current activities before being deallocated. A `Synthesizer` must be in the `QUEUE_EMPTY` state before being deallocated. If the queue is not empty, any objects on the speech output queue must be cancelled with appropriate events issued. A `Recognizer` cannot be in the `PROCESSING` state when being deallocated. If necessary, there must be a `forceFinalize` of any unfinalized result.

While this method is being processed events are issued to any `EngineListeners` attached to the `Engine` to indicate state changes. First, as the `Engine` changes from the `ALLOCATED` to the `DEALLOCATING_RESOURCES` state, an `ENGINE_DEALLOCATING_RESOURCES` event is issued. As the deallocation process completes, the engine moves from the `DEALLOCATING_RESOURCES` state to the `DEALLOCATED` state and an `ENGINE_DEALLOCATED` event is issued.

The `deallocate` method should only be called for an `Engine` in the `ALLOCATED` state. The method has no effect for an `Engine` is either the `DEALLOCATING_RESOURCES` or `DEALLOCATED` states. The method throws an exception in the `ALLOCATING_RESOURCES` state.

Deallocating resources for an engine is not always immediate. Since the deallocate method does not return until complete, applications may want to perform deallocation in a separate thread. The documentation for the `allocate` method shows an example of an inner class implementation of `Runnable` that creates a separate thread.

Throws:
EngineException - if a deallocation error occurs
EngineStateError - if called for an engine in the `ALLOCATING_RESOURCES` state

See Also:
allocate, ENGINE_DEALLOCATED, QUEUE_EMPTY

---

pause

```
public void pause()
         throws EngineStateError
```

Pause the audio stream for the engine and put the `Engine` into the `PAUSED` state. Pausing an engine pauses the underlying engine for *all applications* that are connected to that engine. Engines are typically paused and resumed by request from a user.

Applications may pause an engine indefinately. When an engine moves from the `RESUMED` state to the `PAUSED` state, an `ENGINE_PAUSED` event is issued to each `EngineListener` attached to the `Engine`. The `PAUSED` bit of the engine state is set to `true` when paused, and can be tested by the `getEngineState` method and other engine state methods.

The `PAUSED` state is a sub-state of the `ALLOCATED` state. An `ALLOCATED` `Engine` is always in either the `PAUSED` or the `RESUMED` state.

It is not an exception to pause an `Engine` that is already paused.

The `pause` method operates as defined for engines in the `ALLOCATED` state. When pause is called for an engine in the `ALLOCATING_RESOURCES` state, the method blocks (waits) until the `ALLOCATED` state is reached and then operates normally. An error is thrown when pause is called for an engine is either the `DEALLOCATED` is `DEALLOCATING_RESOURCES` states. state.

The `pause` method does not always return immediately. Some applications need to execute pause in a separate thread. The documentation for the `allocate` method includes an example implementation of `Runnable` with inner classes that can perform `pause` in a separate thread.

Pausing a `Synthesizer`

The pause/resume mechanism for a synthesizer is analogous to pause/resume on a tape player or CD player. The audio output stream is paused. The speaking queue is left intact and a subsequent resume continues output from the point at which the pause took effect.

Pausing a `Recognizer`

Pause and resume for a recognizer are analogous to turning a microphone off and on. Pausing stops the input audio input stream as close as possible to the time of the call to pause. The incoming audio between the pause and the resume calls is *ignored*.

Anything a user says while the recognizer is paused will not be heard by the recognizer. Pausing a recognizer during the middle of user speech forces the recognizer to finalize or reject processing of that incoming speech - a recognition result cannot cross a pause/resume boundary.

Most recognizers have some amount of internal audio buffering. This means that some recognizer processing may continue after the pause. For example, results can be created and finalized.

Note: recognizers add a special `suspend` method that allows applications to temporarily stop the recognizer to modify grammars and grammar activation. Unlike a paused recognizer, a suspended recognizer buffers incoming audio input to be processed once it returns to a listening state, so no audio is lost.

Throws:
EngineStateError - if called for an engine in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
resume, getEngineState, ENGINE_PAUSED, suspend

---

resume

```
public void resume()
           throws AudioException,
                  EngineStateError
```

Put the `Engine` in the `RESUMED` state to resume audio streaming to or from a paused engine. Resuming an engine resuming the underlying engine for *all applications* that are connected to that engine. Engines are typically paused and resumed by request from a user.

The specific pause/resume behavior of recognizers and synthesizers is defined in the documentation for the `pause` method.

When an engine moves from the `PAUSED` state to the `RESUMED` state, an `ENGINE_RESUMED` event is issued to each `EngineListener` attached to the `Engine`. The `RESUMED` bit of the engine state is set to `true` when resumed, and can be tested by the `getEngineState` method and other engine state methods.

The `RESUMED` state is a sub-state of the `ALLOCATED` state. An `ALLOCATED` `Engine` is always in either the `PAUSED` or the `RESUMED` state.

It is not an exception to resume a `engine` that is already in the `RESUMED` state. An exception may be thrown if the audio resource required by the engine (audio input or output) is not available.

The `resume` method operates as defined for engines in the `ALLOCATED` state. When resume is called for an engine in the `ALLOCATING_RESOURCES` state, the method blocks (waits) until the `ALLOCATED` state is reached and then operates normally. An error is thrown when resume is called for an engine is either the `DEALLOCATED` is `DEALLOCATING_RESOURCES` states. state.

The `resume` method does not always return immediately. Some applications need to execute resume in a separate thread. The documentation for the `allocate` method includes an example implementation of `Runnable` with inner classes that could also perform `resume` in a separate thread.

Throws:
AudioException - if unable to gain access to the audio channel
EngineStateError - if called for an engine in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
pause, getEngineState, ENGINE_RESUMED

---

getAudioManager

```
public AudioManager getAudioManager()
```

Return an object which provides management of the audio input or output for the `Engine`.

The `AudioManager` is available in any state of an `Engine`.

Returns:
the `AudioManager` for the engine

---

getVocabManager

```
public VocabManager getVocabManager()
                                       throws EngineStateError
```

Return an object which provides management of the vocabulary for the `Engine`. See the `VocabManager` documentation for a description of vocabularies and their use with speech engines. Returns `null` if the `Engine` does not provide vocabulary management capabilities.

The `VocabManager` is available for engines in the `ALLOCATED` state. The call blocks for engines in the `ALLOCATING_RESOURCES`. An error is thrown for engines in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Returns:
the `VocabManager` for the engine or `null` if it does not have a `VocabManager`
Throws:
EngineStateError - if called for an engine in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
Word

---

getEngineProperties

```
public EngineProperties getEngineProperties()
```

Return the `EngineProperties` object (a JavaBean).

A `Recognizer` returns a `RecognizerProperties` object. The `Recognizer` interface also defines a `getRecognizerProperties` method that returns the same object as `getEngineProperties`, but without requiring a cast to be useful.

A `Synthesizer` returns a `SynthesizerProperties` object. The `Synthesizer` interface also defines a `getSynthesizerProperties` method that returns the same object as `getEngineProperties`, but without requiring a cast to be useful.

The `EngineProperties` are available in any state of an `Engine`. However, changes only take effect once an engine reaches the `ALLOCATED` state.

Returns:
the `EngineProperties` object for this engine
See Also:
getRecognizerProperties, RecognizerProperties, getSynthesizerProperties, SynthesizerProperties

---

getEngineModeDesc

```
public EngineModeDesc getEngineModeDesc()
                                         throws java.lang.SecurityException
```

Return a mode descriptor that defines the operating properties of the engine. For a `Recognizer` the return value is a `RecognizerModeDesc`. For a `Synthesizer` the return value is a `SynthesizerModeDesc`.

The `EngineModeDesc` is available in any state of an `Engine`.

Returns:
an `EngineModeDesc` for the engine.
Throws:
java.lang.SecurityException - if the application does not have `accessEngineModeDesc` permission

---

addEngineListener

```
public void addEngineListener(EngineListener listener)
```

Request notifications of events of related to the `Engine`. An application can attach multiple listeners to an `Engine`. A single listener can be attached to multiple engines.

The `EngineListener` is extended for both recognition and synthesis. Typically, a `RecognizerListener` is attached to a `Recognizer` and a `SynthesizerListener` is attached to a `Synthesizer`.

An `EngineListener` can be attached or removed in any state of an `Engine`.

Parameters:
`listener` - the listener that will receive `EngineEvents`
See Also:
Recognizer, RecognizerListener, Synthesizer, SynthesizerListener

---

removeEngineListener

```
public void removeEngineListener(EngineListener listener)
```

Remove a listener from this `Engine`. An `EngineListener` can be attached or removed in any state of an `Engine`.

Parameters:

`listener` - the listener to be removed

Class javax.speech.EngineAdapter

```
java.lang.Object
  |
  +--javax.speech.EngineAdapter
```

Subclasses:

RecognizerAdapter, SynthesizerAdapter

---

public class **EngineAdapter**
extends java.lang.Object
implements EngineListener

Trivial implementation of the `EngineListener` interface that receives a `EngineEvents`. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Extended by `RecognizerAdapter` and `SynthesizerAdapter`.

See Also:

RecognizerAdapter, SynthesizerAdapter

Constructor Summary

EngineAdapter()	

Method Summary

void	**engineAllocated**(EngineEvent e) The `Engine` has been allocated.
void	**engineAllocatingResources**(EngineEvent e) The `Engine` is being allocated.
void	**engineDeallocated**(EngineEvent e) The `Engine` has been deallocated.
void	**engineDeallocatingResources**(EngineEvent e) The `Engine` is being deallocated.
void	**engineError**(EngineErrorEvent e) An `EngineErrorEvent` has occurred and the `Engine` is unable to continue normal operation.
void	**enginePaused**(EngineEvent e) The `Engine` has been paused.
void	**engineResumed**(EngineEvent e) The `Engine` has been resumed.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

EngineAdapter

`public EngineAdapter()`

Method Detail

enginePaused

`public void enginePaused(EngineEvent e)`

The `Engine` has been paused.

Specified by:
enginePaused in interface EngineListener
See Also:
ENGINE_PAUSED

engineResumed

`public void engineResumed(EngineEvent e)`

The `Engine` has been resumed.

Specified by:
engineResumed in interface EngineListener
See Also:
ENGINE_RESUMED

engineAllocated

`public void engineAllocated(EngineEvent e)`

The `Engine` has been allocated.

Specified by:
engineAllocated in interface EngineListener
See Also:
ENGINE_ALLOCATED

engineDeallocated

`public void engineDeallocated(EngineEvent e)`

The `Engine` has been deallocated.

Specified by:
engineDeallocated in interface EngineListener
See Also:
ENGINE_DEALLOCATED

engineAllocatingResources

`public void engineAllocatingResources(EngineEvent e)`

The `Engine` is being allocated.

Specified by:
engineAllocatingResources in interface EngineListener
See Also:
ENGINE_ALLOCATING_RESOURCES

engineDeallocatingResources

`public void engineDeallocatingResources(EngineEvent e)`

The `Engine` is being deallocated.

Specified by:
engineDeallocatingResources in interface EngineListener
See Also:
ENGINE_DEALLOCATING_RESOURCES

engineError

`public void engineError(EngineErrorEvent e)`

An `EngineErrorEvent` has occurred and the `Engine` is unable to continue normal operation.

Specified by:
engineError in interface EngineListener

Interface javax.speech.EngineCentral

public abstract interface **EngineCentral**

Provides a list of `EngineModeDesc` objects that define the available operating modes of a speech engine.

Each speech engine registers an `EngineCentral` object with the `javax.speech.Central` class. When requested by the `Central` class, each registered `EngineCentral` object provides a list with an `EngineModeDesc` object that describes each available operating mode of the engine.

The `EngineModeDesc` objects returned by `EngineCentral` in its list must implement the `EngineCreate` interface and be a sub-class of either `RecognizerModeDesc` or `SynthesizerModeDesc`. The `Central` class calls the `createEngine` method of the `EngineCentral` interface when it is requested to create an engine. (See `EngineCreate` for more detail.)

The engine must perform the same security checks on access to speech engines as the `Central` class.

Note: Application developers do not need to use this interface. `EngineCentral` is used internally by `Central` and speech engines.

See Also:
Central, EngineCreate, EngineModeDesc

Method Summary

EngineList	**createEngineList**(EngineModeDesc require) Create an `EngineList` containing an `EngineModeDesc` for each mode of operation of a speech engine that matches a set of required features.

Method Detail

createEngineList

```
public EngineList createEngineList(EngineModeDesc require)
                                throws java.lang.SecurityException
```

Create an `EngineList` containing an `EngineModeDesc` for each mode of operation of a speech engine that matches a set of required features. Each object in the list must be a sub-class of either `RecognizerModeDesc` or `SynthesizerModeDesc` and must implement the `EngineCreate` interface.

The `Central` class ensures that the `require` parameter is an instance of either `RecognizerModeDesc` or `SynthesizerModeDesc`. This enables the `EngineCentral` to optimize its search for either recognizers or synthesizers.

Returns `null` if no engines are available or if none meet the specified requirements.

The returned list should indicate the list of modes available at the time of the call (the list may change over time). The engine can create the list at the time of the call or it may be pre-stored.

Throws:
java.lang.SecurityException - if the caller does not have `accessEngineModeDesc` permission

See Also:
EngineCreate, EngineModeDesc, RecognizerModeDesc, SynthesizerModeDesc

Interface javax.speech.EngineCreate

public abstract interface **EngineCreate**

The `EngineCreate` interface is implemented by `EngineModeDesc` objects obtained through calls to the `EngineCentral` objects of each speech engine registered with the `Central` class.

Note: most applications do not need to use this interface.

Each engine implementation must sub-class either `RecognizerModeDesc` or `SynthesizerModeDesc` to implement the `EngineCreate` interface. For example:

```
public MyRecognizerModeDesc extends RecognizerModeDesc implements EngineCreate
...
  public Engine createEngine() {
    // Use mode desc propoerties to create an appropriate engine
  }
```

This implementation mechanism allows the engine to embed additional mode information (engine-specific mode identifiers, GUIDs etc) that simplify creation of the engine if requested by the `Central` class. The engine-specific mode descriptor may need to override equals and other methods if engine-specific features are defined.

The engine must perform the same security checks on access to speech engines as the `Central` class.

See Also:
Central, EngineCentral, EngineModeDesc, RecognizerModeDesc, SynthesizerModeDesc

Method Summary

Engine	**createEngine**() Create an engine with the properties specified by this object.

Method Detail

createEngine

```
public Engine createEngine()
                              throws java.lang.IllegalArgumentException,
                                     EngineException,
                                     java.lang.SecurityException
```

Create an engine with the properties specified by this object. A new engine should be created in the `DEALLOCATED` state.

Throws:
java.lang.IllegalArgumentException - The properties of the `EngineModeDesc` do not refer to a known engine or engine mode.

EngineException - The engine defined by this `EngineModeDesc` could not be properly created.
java.lang.SecurityException - if the caller does not have `createRecognizer` permission but is attempting to create a `Recognizer`

See Also:
DEALLOCATED

Class javax.speech.EngineErrorEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.EngineEvent
                    |
                    +--javax.speech.EngineErrorEvent
```

---

public class **EngineErrorEvent**
extends EngineEvent

`EngineErrorEvent` is an asynchronous notification of an internal error in the engine which prevents normal behavior of that engine. The event encapsulates a `Throwable` object that provides details about the error.

See Also:
engineError, Serialized Form

Field Summary

static int	**ENGINE_ERROR** Identifier for event issued when engine error occurs.
java.lang.Throwable	**problem** `Throwable` object (`Exception` or `Error`) that describes the detected engine problem.

Fields inherited from class javax.speech.EngineEvent
ENGINE_ALLOCATED, ENGINE_ALLOCATING_RESOURCES, ENGINE_DEALLOCATED, ENGINE_DEALLOCATING_RESOURCES, ENGINE_PAUSED, ENGINE_RESUMED, newEngineState, oldEngineState

Fields inherited from class javax.speech.SpeechEvent
id

Fields inherited from class java.util.EventObject
source

Constructor Summary

EngineErrorEvent(Engine source, int id, java.lang.Throwable throwable, long oldEngineState, long newEngineState) Constructs an `EngineErrorEvent` with an event identifier, throwable, old engine state and new engine state.

Method Summary

java.lang.Throwable	**getEngineError**() Return the `Throwable` object (`Exception` or `Error`) that describes the engine problem.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.EngineEvent
getNewEngineState, getOldEngineState, paramString

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

ENGINE_ERROR

`public static final int **ENGINE_ERROR**`

Identifier for event issued when engine error occurs.

See Also:
engineError

---

problem

`protected java.lang.Throwable problem`

`Throwable` object (`Exception` or `Error`) that describes the detected engine problem.

See Also:
getEngineError

Constructor Detail

EngineErrorEvent

```
public EngineErrorEvent(Engine source,
                        int id,
                        java.lang.Throwable throwable,
                        long oldEngineState,
                        long newEngineState)
```

Constructs an `EngineErrorEvent` with an event identifier, throwable, old engine state and new engine state. The old and new states are zero if the engine states are unknown or undefined.

Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type
`throwable` - description of the detected error
`oldEngineState` - engine state prior to this event
`newEngineState` - engine state following this event
See Also:
getEngineState

Method Detail

getEngineError

`public java.lang.Throwable getEngineError()`

Return the `Throwable` object (`Exception` or `Error`) that describes the engine problem.

---

paramString

`public java.lang.String paramString()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class EngineEvent

Class javax.speech.EngineEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.EngineEvent
```

Subclasses:
EngineErrorEvent, RecognizerEvent, SynthesizerEvent

---

public class **EngineEvent**
extends SpeechEvent

`EngineEvent` notifies changes in state of a speech synthesis or recognition engine. `EngineEvents` are issued to each `EngineListener` attached to an engine. The `RecognizerEvent` and `SynthesizerEvent` classes both extend `EngineEvent` to provide specific events for recognizers and synthesizers.

See Also:
EngineListener, RecognizerEvent, SynthesizerEvent, Serialized Form

---

Field Summary

static int	**ENGINE_ALLOCATED** Identifier for event issued when engine allocation is complete.
static int	**ENGINE_ALLOCATING_RESOURCES** Identifier for event issued when engine allocation has commenced.
static int	**ENGINE_DEALLOCATED** Identifier for event issued when engine deallocation is complete.
static int	**ENGINE_DEALLOCATING_RESOURCES** Identifier for event issued when engine deallocation has commenced.
static int	**ENGINE_PAUSED** Identifier for event issued when engine is paused.
static int	**ENGINE_RESUMED** Identifier for event issued when engine is resumed.
long	**newEngineState** Engine state following this event.
long	**oldEngineState** Engine state following prior to this event.

Fields inherited from class javax.speech.SpeechEvent
id

Fields inherited from class java.util.EventObject
source

Constructor Summary

EngineEvent(Engine source, int id, long oldEngineState, long newEngineState) Constructs an `EngineEvent` with an event identifier, old engine state and new engine state.

Method Summary

long	**getNewEngineState**() Return the state following this `EngineEvent`.
long	**getOldEngineState**() Return the state prior to this `EngineEvent`.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

ENGINE_ALLOCATED

`public static final int ENGINE_ALLOCATED`

Identifier for event issued when engine allocation is complete. The `ALLOCATED` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, allocate, engineAllocated

ENGINE_DEALLOCATED

`public static final int ENGINE_DEALLOCATED`

Identifier for event issued when engine deallocation is complete. The `DEALLOCATED` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, allocate, engineDeallocated

ENGINE_ALLOCATING_RESOURCES

`public static final int ENGINE_ALLOCATING_RESOURCES`

Identifier for event issued when engine allocation has commenced. The `ALLOCATING_RESOURCES` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, allocate, engineAllocatingResources

ENGINE_DEALLOCATING_RESOURCES

`public static final int ENGINE_DEALLOCATING_RESOURCES`

Identifier for event issued when engine deallocation has commenced. The `DEALLOCATING_RESOURCES` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, allocate, engineDeallocatingResources

ENGINE_PAUSED

`public static final int ENGINE_PAUSED`

Identifier for event issued when engine is paused. The `PAUSED` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, pause, enginePaused

ENGINE_RESUMED

`public static final int ENGINE_RESUMED`

Identifier for event issued when engine is resumed. The `RESUMED` flag of the `newEngineState` is set.

See Also:
getNewEngineState, getId, resume, engineResumed

newEngineState

`protected long newEngineState`

Engine state following this event.

See Also:
getNewEngineState

oldEngineState

`protected long oldEngineState`

Engine state following prior to this event.

See Also:
getOldEngineState

Constructor Detail

EngineEvent

```
public EngineEvent(Engine source,
                   int id,
                   long oldEngineState,
                   long newEngineState)
```

Constructs an `EngineEvent` with an event identifier, old engine state and new engine state.

Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type
`oldEngineState` - engine state prior to this event
`newEngineState` - engine state following this event

See Also:
getEngineState

Method Detail

getNewEngineState

`public long **getNewEngineState**()`

Return the state following this `EngineEvent`. The value matches the `getEngineState` method.

See Also:
getEngineState

---

getOldEngineState

`public long **getOldEngineState**()`

Return the state prior to this `EngineEvent`. The value matches the `getEngineState` method.

See Also:
getEngineState

---

paramString

`public java.lang.String **paramString**()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class SpeechEvent

Class javax.speech.EngineException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
                    |
                    +--javax.speech.EngineException
```

public class **EngineException**
extends SpeechException

Signals that an error occurred while trying to create or access a speech synthesis engine, speech recognition engine or `EngineCentral` object.

See Also:
Serialized Form

Constructor Summary
EngineException() Construct an `EngineException` with no detail message.
EngineException(java.lang.String s) Construct an `EngineException` with a detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

EngineException

`public **EngineException**()`

Construct an `EngineException` with no detail message.

EngineException

`public **EngineException**(java.lang.String s)`

Construct an `EngineException` with a detail message. A detail message is a `String` that describes this particular exception.

Class javax.speech.EngineList

```
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.Vector
                    |
                    +--javax.speech.EngineList
```

public class **EngineList**
extends java.util.Vector

`EngineList` is a container for a set of `EngineModeDesc` objects. An `EngineList` is used in the selection of speech engines in conjuction with the methods of the `Central` class. It provides convenience methods for the purpose of testing and manipulating the `EngineModeDesc` objects it contains.

An `EngineList` object is typically obtained through the `availableSynthesizers` or `availableRecognizers` methods of the `javax.speech.Central` class. The `orderByMatch`, `anyMatch`, `requireMatch` and `rejectMatch` methods are used to prune the list to find the best match given multiple criteria.

See Also:
EngineModeDesc, Central, availableRecognizers, availableSynthesizers, Serialized Form

Fields inherited from class java.util.Vector	
capacityIncrement, elementCount, elementData	

Fields inherited from class java.util.AbstractList	
modCount	

Method Summary

boolean	**anyMatch**(EngineModeDesc require) Return true if one or more `EngineModeDesc` in the `EngineList` match the required properties.
void	**orderByMatch**(EngineModeDesc require) Order the list so that elements matching the required features are at the head of the list, and others are at the end.
void	**rejectMatch**(EngineModeDesc reject) Remove `EngineModeDesc` entries from the list that do match `require`.
void	**requireMatch**(EngineModeDesc require) Remove `EngineModeDesc` entries from the list that do not match `require`.

Methods inherited from class java.util.Vector
addAll, addAll, addElement, add, add, capacity, clear, clone, containsAll, contains, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, removeAllElements, removeAll, removeElementAt, removeElement, remove, remove, retainAll, setElementAt, setSize, set, size, toArray, toArray, toString, trimToSize

Methods inherited from class java.util.AbstractList
add, add, addAll, equals, get, hashCode, indexOf, iterator, lastIndexOf, listIterator, listIterator, remove, set, subList

Methods inherited from class java.util.AbstractCollection
add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

anyMatch

`public boolean **anyMatch**(EngineModeDesc require)`

Return true if one or more `EngineModeDesc` in the `EngineList` match the required properties. The `require` object is tested with the `match` method of each `EngineModeDesc` in the list. If any match call returns true then this method returns `true`.

`anyMatch` is often used to test whether pruning a list (with `requireMatch` or `rejectMatch`) would leave the list empty.

See Also:
match

---

requireMatch

`public void **requireMatch**(EngineModeDesc require)`

Remove `EngineModeDesc` entries from the list that do not match `require`. The `match` method for each `EngineModeDesc` in the list is called: if it returns `false` it is removed from the list. Example:

```
// Remove engine mode descriptors from a list if they
// don’t support US English.
EngineList list = ....;
EngineModeDesc desc = new EngineModeDesc(Locale.US);
list.requireMatch(desc);
```

See Also:
match

---

rejectMatch

`public void **rejectMatch**(EngineModeDesc reject)`

Remove `EngineModeDesc` entries from the list that do match `require`. The `match` method for each `EngineModeDesc` in the list is called: if it returns `true` it is removed from the list. Example:

```
// Remove engine mode descriptors if they support US English.
EngineList list = ....;
EngineModeDesc desc = new EngineModeDesc(Locale.US);
list.rejectMatch(desc);
```

See Also:
match

---

orderByMatch

`public void **orderByMatch**(EngineModeDesc require)`

Order the list so that elements matching the required features are at the head of the list, and others are at the end. Within categories, the original order of the list is preserved. Example:

```
// Put running engines at the head of the list.
EngineList list = ....;
EngineModeDesc desc = new EngineModeDesc();
desc.setRunning(true);
list.orderByMatch(desc);
```

Interface javax.speech.EngineListener

Subinterfaces:
RecognizerListener, SynthesizerListener

Implementing Classes:
EngineAdapter

public abstract interface **EngineListener**
extends java.util.EventListener

Interface defining methods to be called when state-change events for a speech engine occur. To receive engine events an application attaches a listener by calling the `addEngineListener` method of an `Engine`. A listener is removed by a call to the `removeEngineListener` method.

The event dispatch policy is defined in the documentation for the `SpeechEvent` class.

This interface is extended by the `RecognizerListener` and `SynthesizerListener` interfaces to handle the specialized events of speech recognizers and synthesizers.

A trivial implementation of `EngineListener` is provided by the `EngineAdapter` class. `RecognizerAdapter` and `SynthesizerAdapter` classes are also available.

See Also:
SpeechEvent, EngineAdapter, RecognizerListener, RecognizerAdapter, SynthesizerListener, SynthesizerAdapter

Method Summary

void	**engineAllocated**(EngineEvent e) The `Engine` has been allocated.
void	**engineAllocatingResources**(EngineEvent e) The `Engine` is being allocated.
void	**engineDeallocated**(EngineEvent e) The `Engine` has been deallocated.
void	**engineDeallocatingResources**(EngineEvent e) The `Engine` is being deallocated.
void	**engineError**(EngineErrorEvent e) An `EngineErrorEvent` has occurred and the `Engine` is unable to continue normal operation.
void	**enginePaused**(EngineEvent e) The `Engine` has been paused.
void	**engineResumed**(EngineEvent e) The `Engine` has been resumed.

Method Detail

enginePaused

`public void` **`enginePaused`**`(EngineEvent e)`

The `Engine` has been paused.

See Also:
ENGINE_PAUSED

---

engineResumed

`public void` **`engineResumed`**`(EngineEvent e)`

The `Engine` has been resumed.

See Also:
ENGINE_RESUMED

---

engineAllocated

`public void` **`engineAllocated`**`(EngineEvent e)`

The `Engine` has been allocated.

See Also:
ENGINE_ALLOCATED

---

engineDeallocated

`public void` **`engineDeallocated`**`(EngineEvent e)`

The `Engine` has been deallocated.

See Also:
ENGINE_DEALLOCATED

---

engineAllocatingResources

`public void` **`engineAllocatingResources`**`(EngineEvent e)`

The `Engine` is being allocated.

See Also:
ENGINE_ALLOCATING_RESOURCES

---

engineDeallocatingResources

`public void` **`engineDeallocatingResources`**`(EngineEvent e)`

The `Engine` is being deallocated.

See Also:
ENGINE_DEALLOCATING_RESOURCES

---

engineError

`public void` **`engineError`**`(EngineErrorEvent e)`

An `EngineErrorEvent` has occurred and the `Engine` is unable to continue normal operation.

See Also:
EngineErrorEvent

Class javax.speech.EngineModeDesc

```
java.lang.Object
  |
  +--javax.speech.EngineModeDesc
```

Subclasses:
RecognizerModeDesc, SynthesizerModeDesc

---

public class **EngineModeDesc**
extends java.lang.Object

`EngineModeDesc` provides information about a *specific operating mode* of a speech engine. The `availableRecognizers` and `availableSynthesizers` methods of the `Central` class provide a list of mode descriptors for all operating modes of registered engines. Applications may also create `EngineModeDescs` for use in selecting and creating engines. Examples of uses mode descriptors are provided in the documentation for the `Central` class.

The properties defined in the `EngineModeDesc` class apply to all speech engines including speech recognizers and speech synthesizers. The `RecognizerModeDesc` and `SynthesizerModeDesc` classes extend the `EngineModeDesc` class to define specialized properties for recognizers and synthesizers.

The `EngineModeDesc` and its sub-classes follow the Java Beans set/get property patterns. The list of properties is outlined below.

The properties of `EngineModeDesc` and its sub-classes are all object references. All properties are defined so that a `null` value means "don't care" when selecting an engine or matching `EngineModeDesc` and its sub-classes. For example, a `Boolean` value for a property means that its three values are true, false and don't care (`null`).

The basic properties of an engine defined by `EngineModeDesc` are:

engine name
A string that uniquely identifies a speech engine. e.g. "Acme Recognizer"
mode name
A string that uniquely identifies a mode of operation of the speech engine. e.g. "Spanish Dictator"
Locale
A `java.util.Locale` object representing the language supported by the engine mode. The country code may be optionally defined for an engine. The Locale variant is typically ignored. e.g. Locale("fr", "CA") represent French spoken in Canada. ("fr" and "CA" are standard ISO codes).
Running
A `Boolean` value indicating whether a speech engine is already running. This allows for the selection of engines that already running so that system resources are conserved.

Selection

There are two types of `EngineModeDesc` object (and its sub-classes): those created by a speech engine and those created by an application. Engine-created descriptors are obtained through the `availableRecognizers` and `availableSynthesizers` methods of the `Central` class and must have all features set to non-null values.

Applications can create descriptors using the constructors of the descriptor classes. Applications may leave any or all of the feature values `null` to indicate "don't care".

Typically, application-created descriptors are used to test the engine-created descriptors to select an appropriate engine for creation. For example, the following code tests whether an engine mode supports Swiss German:

```
EngineModeDesc fromEngine = ...;
// "de" is the ISO 639 language code for German
// "CH" is the ISO 3166 country code for Switzerland
// (see locale for details)
EngineModeDesc require = new EngineModeDesc(new Locale("de", "CH"));
// test whether the engine mode supports Swiss German.
if (fromEngine.match(require)) ...
```

An application can create a descriptor and pass it to the `createRecognizer` or `createSynthesizer` methods of `Central`. In this common approach, the `Central` performs the engine selection.

```
// Create a mode descriptor that requires French
EngineModeDesc desc = new EngineModeDesc(Locale.FRENCH);
// Create a synthesizer that supports French
Synthesizer synth = Central.createSynthesizer(desc);
```

Applications that need advanced selection criterion will

1. Request a list of engine mode descriptors from `availableRecognizers` or `availableSynthesizers`,
2. Select one of the descriptors using the methods of `EngineList` and `EngineModeDesc` and its sub-classes,
3. Pass the selected descriptor to the `createRecognizer` or `createSynthesizer` method of `Central`.

See Also:
RecognizerModeDesc, SynthesizerModeDesc, Central

Constructor Summary
EngineModeDesc() Empty constructor sets engine name, mode name, Locale and running all to `null`.
EngineModeDesc(java.util.Locale locale) Construct an `EngineModeDesc` for a locale.
EngineModeDesc(java.lang.String engineName, java.lang.String modeName, java.util.Locale locale, java.lang.Boolean running) Constructor provided with engine name, mode name, locale and running.

Method Summary

boolean	**equals**(java.lang.Object anObject) True if and only if the parameter is not `null` and is a `EngineModeDesc` with equal values of Locale, engineName and modeName.
java.lang.String	**getEngineName**() Get the engine name.
java.util.Locale	**getLocale**() Get the Locale.
java.lang.String	**getModeName**() Get the mode name.
java.lang.Boolean	**getRunning**() Get the running feature.
boolean	**match**(EngineModeDesc require) Determine whether an `EngineModeDesc` has all the features defined in the `require` object.
void	**setEngineName**(java.lang.String engineName) Set the engine name.
void	**setLocale**(java.util.Locale locale) Set the Locale.
void	**setModeName**(java.lang.String modeName) Set the mode name.
void	**setRunning**(java.lang.Boolean running) Set the running feature.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

EngineModeDesc

```
public EngineModeDesc()
```

Empty constructor sets engine name, mode name, Locale and running all to `null`.

EngineModeDesc

```
public EngineModeDesc(java.util.Locale locale)
```

Construct an `EngineModeDesc` for a locale. The engine name, mode name and running are set to `null`.

EngineModeDesc

```
public EngineModeDesc(java.lang.String engineName,
                      java.lang.String modeName,
                      java.util.Locale locale,
                      java.lang.Boolean running)
```

Constructor provided with engine name, mode name, locale and running. Any parameter may be `null`.

Method Detail

getEngineName

```
public java.lang.String getEngineName()
```

Get the engine name. The engine name should be a unique string across the provider company and across companies.

---

setEngineName

```
public void setEngineName(java.lang.String engineName)
```

Set the engine name. May be null.

---

getModeName

```
public java.lang.String getModeName()
```

Get the mode name. The mode name that should uniquely identify a single mode of operation of a speech engine (per-engine unique).

---

setModeName

```
public void setModeName(java.lang.String modeName)
```

Set the mode name. May be null.

---

getLocale

`public java.util.Locale **getLocale**()`

Get the Locale. The locale for an engine mode must have the language defined but the country may be undefined. (e.g. `Locale.ENGLISH` indicates the English language spoken in any country). The locale variant is typically ignored.

setLocale

`public void **setLocale**(java.util.Locale locale)`

Set the Locale. May be null.

getRunning

`public java.lang.Boolean **getRunning**()`

Get the running feature. Values may be `TRUE`, `FALSE` or `null` (null means "don't care").

setRunning

`public void **setRunning**(java.lang.Boolean running)`

Set the running feature. Values may be `TRUE`, `FALSE` or `null` (null means "don't care").

match

`public boolean **match**(EngineModeDesc require)`

Determine whether an `EngineModeDesc` has all the features defined in the `require` object. Strings in `require` which are either `null` or zero-length ("") are not tested, including those in the Locale. All string comparisons are exact (case-sensitive).

equals

`public boolean **equals**(java.lang.Object anObject)`

True if and only if the parameter is not `null` and is a `EngineModeDesc` with equal values of Locale, engineName and modeName.
Overrides:
equals in class java.lang.Object

Interface javax.speech.EngineProperties

Subinterfaces:
RecognizerProperties, SynthesizerProperties

public abstract interface **EngineProperties**

An `EngineProperties` object defines the set of run-time properties of an `Engine`. This interface is extended for each type of speech engine. `SynthesizerProperties` and `RecognizerProperties` define the additional run-time properties of synthesizers and recognizers respectively. The `EngineProperties` object for an `Engine` is obtained from the `getEngineProperties` method of the engine, and should be cast appropriately for the type of engine. For example:

```
Synthesizer synth = ...;
SynthesizerProperties props = (SynthesizerProperties)synth.getEngineProperties();
```

Each property of an engine has a set and get method. The method signatures follow the JavaBeans design patterns (outlined below).

The run-time properties of an engine affect the behavior of a running engine. Technically, properties affect engines in the `ALLOCATED` state. Normally, property changes are made on an `ALLOCATED` engine and take effect immediately or soon after the change call.

The `EngineProperties` object for an engine is, however, available in all states of an `Engine`. Changes made to the properties of engine in the `DEALLOCATED` or the `ALLOCATING_RESOURCES` state take effect when the engine next enters the `ALLOCATED` state. A typical scenario for setting the properties of a non-allocated is determining the initial state of the engine. For example, setting the initial voice of a `Synthesizer`, or the initial `SpeakerProfile` of a `Recognizer`. (Setting these properties prior to allocation is desirable because allocating the engine and then changing the voice or the speaker can be computationally expensive.)

When setting any engine property:

- The engine may choose to ignore a set value either because it does not support changes in a property or because it is out-of-range.
- The engine will apply the property change as soon as possible, but the change is not necessarily immediate. Thus, all set methods are asynchronous (call may return before the effect takes place).
- All properties of an engine are *bound* properties - JavaBeans terminology for a property for which an event is issued when the property changes. A `PropertyChangeListener` can be attached to the `EngineProperties` object to receive a property change event when a change takes effect.

For example, a call to the `setPitch` method of the `SynthesizerProperties` interface to change pitch from 120Hz to 200Hz might fail because the value is out-of-range. If it does succeed, the pitch change might be deferred until the synthesizer can make the change by waiting to the end of the current word, sentence, paragraph or text object. When the change does take effect, a `PropertyChangeEvent` is issued to all attached listeners with the name of the changed property ("Pitch"), the old value (120) and the new value (200).

Set calls take effect in the order in which they are received. If multiple changed are requested are requested for a single property, a separate event is issued for each call, even if the multiple changes take place simultaneously.

The properties of an engine are persistent across sessions where possible. It is the engine's responsibility to store and restore the property settings. In multi-user and client-server environments the store/restore policy is at the discretion of the engine.

Control Component

An engine may provide a control object through the `getControlComponent` method of its `EngineProperties` object. The control object is an AWT `Component`. If provided, that component can be displayed to a user for customization of the engine. Because the component is implemented by the engine, the display may support customization of engine-specific properties that are not accessible through the standard properties interfaces.

JavaBeans Properties

The JavaBeans property patterns are followed for engine properties. A property is defined by its name and its property type (for example, "Pitch" is a float). The property is accessed through get and set methods. The signature of the property accessor methods are:

```
public void set<PropertyName>(<PropertyType> value);
public <PropertyType> get<PropertyName>();
```

For boolean-valued properties, the `get<PropertyName>()` may also be `is<PropertyName>()`

For indexed properties (arrays) the signature of the property accessor methods are:

```
public void set<PropertyName>(<PropertyType>[]; value);
public void set<PropertyName>(int i, <PropertyType> value);
public <PropertyType>[] get<PropertyName>();
public <PropertyType> get<PropertyName>(int i);
```

For example speaking rate (for a Synthesizer) is a floating value and has the following methods:

```
public void setSpeakingRate(float value);
public float getSpeakingRate();
```

See Also:
Engine, getEngineProperties, RecognizerProperties, SynthesizerProperties, java.beans.PropertyChangeListener, java.beans.PropertyChangeEvent

---

Method Summary

void	**addPropertyChangeListener**(java.beans.PropertyChangeListener listener) Add a `PropertyChangeListener` to the listener list.
java.awt.Component	**getControlComponent**() Obtain the AWT Component that provides the default user interface for setting the properties of this `Engine`.
void	**removePropertyChangeListener**(java.beans.PropertyChangeListener listener) Remove a `PropertyChangeListener` from the listener list.
void	**reset**() The reset method returns all properties to reasonable defaults for the `Engine`.

Method Detail

getControlComponent

`public java.awt.Component getControlComponent()`

Obtain the AWT Component that provides the default user interface for setting the properties of this `Engine`. If this `Engine` has no default control panel, the return is `null`

---

reset

`public void reset()`

The reset method returns all properties to reasonable defaults for the `Engine`. A property change event is issued for each engine property that changes as the reset takes effect.

---

addPropertyChangeListener

`public void addPropertyChangeListener(java.beans.PropertyChangeListener listener)`

Add a `PropertyChangeListener` to the listener list. The listener is registered for all properties of the engine..

A `PropertyChangeEvent` is fired in response to setting any bound property.
Parameters:
`listener` - - The `PropertyChangeListener` to be added

removePropertyChangeListener

```
public void removePropertyChangeListener(java.beans.PropertyChangeListener listener)
```

Remove a `PropertyChangeListener` from the listener list.

Parameters:

`listener` - - The `PropertyChangeListener` to be removed

Class javax.speech.EngineStateError

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
              |
              +--javax.speech.SpeechError
                    |
                    +--javax.speech.EngineStateError
```

public class **EngineStateError**
extends SpeechError

Signals an error caused by an illegal call to a method of a speech engine. For example, it is illegal to request a deallocated `Synthesizer` to `speak`, or to request a deallocated `Recognizer` to create or new grammar, or to request any deallocated engine to pause or resume.

See Also:

Serialized Form

Constructor Summary

EngineStateError() Construct an `EngineStateError` with no detail message.
EngineStateError(java.lang.String s) Construct an `EngineStateError` with a detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

EngineStateError

`public **EngineStateError**()`

Construct an `EngineStateError` with no detail message.

EngineStateError

`public **EngineStateError**(java.lang.String s)`

Construct an `EngineStateError` with a detail message. A detail message is a `String` that describes this particular exception.

Class javax.speech.SpeechError

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
              |
              +--javax.speech.SpeechError
```

Subclasses:
EngineStateError, ResultStateError

public class **SpeechError**
extends java.lang.Error

Signals that an error has occurred in the javax.speech package. `SpeechError` is the super class of all errors in the Java Speech API.

See Also:
SpeechException, Serialized Form

Constructor Summary

SpeechError(java.lang.String s) Constructs a `SpeechException` with a detail message.
SpeechError() Empty constructor for `SpeechException` with no detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

SpeechError

`public SpeechError(java.lang.String s)`

Constructs a `SpeechException` with a detail message.

Parameters:

`s` - the detail message

SpeechError

`public SpeechError()`

Empty constructor for `SpeechException` with no detail message.

Parameters:

`s` - the detail message

Class javax.speech.SpeechEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
```

Subclasses:

AudioEvent, EngineEvent, GrammarEvent, ResultEvent, SpeakableEvent

public class **SpeechEvent**
extends java.util.EventObject

The root event class for all speech events. All events from a speech engine (recognizer or synthesizer) are synchronized with the AWT event queue. This allows an application to mix speech and AWT events with being concerned with multi-threading problems.

Note to Engine Developers

The AWT event queue is obtained through the AWT Toolkit:

```
import java.awt.*;
...
EventQueue q = Toolkit.getDefaultToolkit().getSystemEventQueue();
```

An engine should create a sub-class of `AWTEvent` that can be placed on the AWT event queue. The engine also needs to create a non-visual AWT `Component` to receive the engine's `AWTEvent`. When the AWT event is notified to the engine's component, the engine should issue the approporiate speech event. The speech event can be issued either from the AWT event thread or from a separate thread created by the speech engine. (Note that `SpeechEvent` is not a sub-class of `AWTEvent` so speech events can not be placed directly onto the AWT event queue.)

See Also:

Serialized Form

Field Summary

int	**id** Event identifier.

Fields inherited from class java.util.EventObject
source

Constructor Summary
SpeechEvent(java.lang.Object source) Constructs a `SpeechEvent` with a specified source.
SpeechEvent(java.lang.Object source, int id) Constructs a `SpeechEvent`.

Method Summary	
int	**getId**() Return the event identifier.
java.lang.String	**paramString**() Returns a parameter string identifying this event.
java.lang.String	**toString**() Return a printable String.

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

id

`protected int id`

Event identifier. Id values are defined for each sub-class of `SpeechEvent`.

See Also:

getId

Constructor Detail

SpeechEvent

`protected SpeechEvent(java.lang.Object source)`

Constructs a `SpeechEvent` with a specified source. The source must be non-null.

SpeechEvent

```
protected SpeechEvent(java.lang.Object source,
                      int id)
```

Constructs a `SpeechEvent`.
Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type

Method Detail

getId

`public int getId()`

Return the event identifier. Id values are defined for each sub-class of `SpeechEvent`.

---

paramString

`public java.lang.String paramString()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event

---

toString

`public java.lang.String toString()`

Return a printable String. Useful for event-logging and debugging.
Overrides:
toString in class java.util.EventObject

Class javax.speech.SpeechException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
```

Subclasses:
AudioException, EngineException, GrammarException, JSMLException, VendorDataException

---

public class **SpeechException**
extends java.lang.Exception

Signals that a Java Speech API exception has occurred. `SpeechException` is the super class of all exceptions in the `javax.speech` packages.

In addition to exceptions that inherit from `SpeechException` some calls throw other Java `Exceptions` and `Errors` such as `IllegalArgumentException` and `SecurityException`.

See Also:
SpeechError, Serialized Form

Constructor Summary
SpeechException() Constructs a `SpeechException` with no detail message.
SpeechException(java.lang.String s) Constructs a `SpeechException` with the specified detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

SpeechException

`public SpeechException()`

Constructs a `SpeechException` with no detail message.

SpeechException

`public SpeechException(java.lang.String s)`

Constructs a `SpeechException` with the specified detail message. A detail message is a `String` that describes this particular exception.
Parameters:
`s` - the detail message

Class javax.speech.SpeechPermission

```
java.lang.Object
  |
  +--java.security.Permission
        |
        +--java.security.BasicPermission
              |
              +--javax.speech.SpeechPermission
```

---

public final class **SpeechPermission**
extends java.security.BasicPermission

This class represents speech permissions. A `SpeechPermission` contains a target name but no actions list; you either have the named permission or you don't.

The target name is the name of the speech permission. The naming convention follows the hierarchical property naming convention. So, an asterisk could be used to represent all speech permissions.

In Java Speech API version 1.0 there is a single `SpeechPermission` - `javax.speech`. When that permission is granted an application or applet has access to all the capabilities provided by installed speech recognizers and synthesizers. Without that permission an application or applet has no access to speech capabilities.

As speech technology matures it is anticipated that a finer-grained permission model will be introduced to provide access by applications and applets to some, but not all, speech capabilities.

Before granting speech permission, developers and users should consider the potential impact of the grant.

Permission Target Name	What the Permission Allows	Risks of Allowing this Permission
`javax.speech`	Creation of a speech recognizer installed on the client.	A malicious applet could use a `RuleGrammar` or `DictationGrammar` to monitor ("bug") a user's office by transmitting the text or audio of results.
	Access to user information stored in speaker profiles possibly including user names and identifiers.	Applet can obtain user names, user identifiers and possibly other sensitive information about users of a system.
	Access to `EngineModeDesc` objects through the the `Central` class and from engines.	Applet could obtain information about the speech technology software installed on a system.

See Also:
java.security.BasicPermission, java.lang.SecurityManager, Serialized Form

Constructor Summary

SpeechPermission(java.lang.String name)
Creates a new `SpeechPermission` with the specified name.

SpeechPermission(java.lang.String name, java.lang.String actions)
Creates a new `SpeechPermission` object with the specified name.

Methods inherited from class java.security.BasicPermission
equals, getActions, hashCode, implies, newPermissionCollection

Methods inherited from class java.security.Permission
checkGuard, equals, getActions, getName, hashCode, implies, newPermissionCollection, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

SpeechPermission

```
public SpeechPermission(java.lang.String name)
```

Creates a new `SpeechPermission` with the specified name. The name is the symbolic name of the `SpeechPermission`: e.g., `javax.speech`.

Parameters:
`name` - the name of the `SpeechPermission`.

SpeechPermission

```
public SpeechPermission(java.lang.String name,
                        java.lang.String actions)
```

Creates a new `SpeechPermission` object with the specified name. The name is the symbolic name of the `SpeechPermission`, and the actions `String` is currently unused and should be `null`. This constructor exists for use by the `Policy` object to instantiate new `Permission` objects.

Parameters:

`name` - the name of the `SpeechPermission`.
`actions` - should be `null`.

Class javax.speech.VendorDataException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
                    |
                    +--javax.speech.VendorDataException
```

public class **VendorDataException**
extends SpeechException

Signals that a problem has been encountered loading or saving some type of vendor-specific data.

See Also:
readVendorGrammar(java.io.InputStream), readVendorResult(java.io.InputStream), readVendorSpeakerProfile(java.io.InputStream), Serialized Form

Constructor Summary

Constructor Summary
VendorDataException() Construct a `VendorDataException` with no detail message.
VendorDataException(java.lang.String s) Construct a `VendorDataException` with the specified detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

VendorDataException

`public **VendorDataException**()`

Construct a `VendorDataException` with no detail message.

VendorDataException

`public **VendorDataException**(java.lang.String s)`

Construct a `VendorDataException` with the specified detail message. A detail message is a `String` that describes this particular exception.
Parameters:
`s` - the detail message

Interface javax.speech.VocabManager

public abstract interface **VocabManager**

Interface for management of words used by a speech `Engine`. The `VocabManager` for an `Engine` is returned by the `getVocabManager` method of the `Engine` interface. Engines are not required support the `VocabManager` - the `getVocabManager` manager may return `null`.

Words, technically known as tokens, are provided to the vocabulary manager with optional information about their pronunciation, grammatical role and spoken form.

The `VocabManager` is typically used to provide a speech engine with information on problematic words - usually words for which the engine is unable to guess a pronunciation. For debugging purposes, an `Engine` may provide a list of words it finds difficult through the `listProblemWords` method.

Words in the vocabulary manager can be used as tokens in rule grammars for recognizers.

See Also:
getVocabManager, Word

Method Summary

void	**addWords**(Word[] w) Add an array of words to the vocabulary.
void	**addWord**(Word w) Add a word to the vocabulary.
Word[]	**getWords**(java.lang.String text) Get all words from the vocabulary manager matching text.
Word[]	**listProblemWords**() Returns a list of problematic words encountered during a session of using a speech recognizer or synthesizer.
void	**removeWords**(Word[] w) Remove an array of words from the vocabulary.
void	**removeWord**(Word w) Remove a word from the vocabulary.

Method Detail

addWord

`public void **addWord**(Word w)`

Add a word to the vocabulary.

---

addWords

`public void addWords(Word[] w)`

Add an array of words to the vocabulary.

---

removeWord

```
public void removeWord(Word w)
                throws java.lang.IllegalArgumentException
```

Remove a word from the vocabulary.
Throws:
java.lang.IllegalArgumentException - Word is not known to the VocabManager.

---

removeWords

```
public void removeWords(Word[] w)
                 throws java.lang.IllegalArgumentException
```

Remove an array of words from the vocabulary. To remove a set of words it is often useful to

```
removeWords(getWords("matching"));
```

Throws:
java.lang.IllegalArgumentException - Word is not known to the VocabManager.

---

getWords

`public Word[] getWords(java.lang.String text)`

Get all words from the vocabulary manager matching text. Returns `null` if there are no matches. If text is `null` all words are returned. This method only returns words that have been added by the `addWord` methods - it does not provide access to the engine's internal word lists.
Parameters:
`text` - word requested from `VocabManager`
Returns:
list of `Words` matching `text`

---

listProblemWords

`public Word[] listProblemWords()`

Returns a list of problematic words encountered during a session of using a speech recognizer or synthesizer. This return information is intended for development use (so the application can be enhanced to provide vocabulary information). An engine may return null.

If a pronunciation for a problem word is provided through the `addWord` methods, the engine may remove the word from the problem list.

An engine may (optionally) include its best-guess pronunciations for problem words in the return array allowing the developer to fix (rather than create) the pronunciation.

Class javax.speech.Word

```
java.lang.Object
  |
  +--javax.speech.Word
```

---

public class **Word**
extends java.lang.Object

The `Word` class provides a standard representation of speakable words for speech engines. A `Word` object provides the following information:

- "Written form" string: text that can be used to present the `Word` visually.
- "Spoken form" text: printable string that indicates how the word is spoken.
- Pronunciation: a string of phonetic characters indicating how the word is spoken.
- Grammatical categories: flags indicating grammatical "part-of-speech" information.

The written form string is required. The other properties are optional. Typically, one or more of the optional properties are specified. The `Word` class allows the specification of multiple pronunciations and multiple grammatical categories. Each pronunciation must be appropoiate to each category. If not, separate `Word` objects should be created.

All the optional properties of a word are hints to the speech engine. Speech engines will use the information as appropriate for their internal design.

See Also:
VocabManager, getVocabManager

Field Summary

static long	**ABBREVIATION** Word is an abbreviation or acronynm.
static long	**ADJECTIVE** Grammatical category of word is adjective.
static long	**ADVERB** Grammatical category of word is adverb.
static long	**AUXILIARY** Grammatical category of word is auxiliary.
static long	**CARDINAL** Grammatical category of word is cardinal.
static long	**CONJUNCTION** Grammatical category of word is conjunction.
static long	**CONTRACTION** Grammatical category is contraction.
static long	**DETERMINER** Grammatical category of word is determiner.
static long	**DONT_CARE** Grammatical category of word doesn’t matter.
static long	**NOUN** Grammatical category of word is noun.
static long	**OTHER** Other grammatical category of word not specified elsewhere in this class.
static long	**PREPOSITION** Grammatical category of word is preposition.
static long	**PRONOUN** Grammatical category of word is pronoun.
static long	**PROPER_ADJECTIVE** Grammatical category of word is proper adjective.
static long	**PROPER_NOUN** Grammatical category of word is proper noun.
static long	**UNKNOWN** Grammatical category of word is unknown.
static long	**VERB** Grammatical category of word is verb.

Method Summary

long	**getCategories**() Get the categories of the `Word`.
java.lang.String[]	**getPronunciations**() Get the pronunciations of the `Word`.
java.lang.String	**getSpokenForm**() Get the "spoken form" of the `Word`.
java.lang.String	**getWrittenForm**() Get the written form of the `Word`.
void	**setCategories**(long cat) Set the categories of the `Word`.
void	**setPronunciations**(java.lang.String[] pron) Set the pronunciation of the `Word` as an array containing a phonetic character `String` for each pronunciation of the word.
void	**setSpokenForm**(java.lang.String text) Set the "spoken form" of the `Word`.
void	**setWrittenForm**(java.lang.String text) Set the "written form" of the `Word`.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

UNKNOWN

`public static final long UNKNOWN`

Grammatical category of word is unknown. The value is zero - `0` - and implies that no other category flag is set.

DONT_CARE

`public static final long DONT_CARE`

Grammatical category of word doesn’t matter.

OTHER

`public static final long OTHER`

Other grammatical category of word not specified elsewhere in this class.

NOUN

`public static final long NOUN`

Grammatical category of word is noun. English examples: "car", "house", "elephant".

PROPER_NOUN

`public static final long PROPER_NOUN`

Grammatical category of word is proper noun. English examples: "Yellowstone", "Singapore".

PRONOUN

`public static final long PRONOUN`

Grammatical category of word is pronoun. English examples: "me", "I", "they".

VERB

`public static final long VERB`

Grammatical category of word is verb. English examples: "run", "debug", "integrate".

ADVERB

`public static final long ADVERB`

Grammatical category of word is adverb. English examples: "slowly", "loudly", "barely", "very", "never".

ADJECTIVE

`public static final long ADJECTIVE`

Grammatical category of word is adjective. English examples: "red", "mighty", "very", "first", "eighteenth".

PROPER_ADJECTIVE

`public static final long` **`PROPER_ADJECTIVE`**

Grammatical category of word is proper adjective. English examples: "British", "Brazilian".

AUXILIARY

`public static final long` **`AUXILIARY`**

Grammatical category of word is auxiliary. English examples: "have", "do", "is", "shall", "must", "cannot".

DETERMINER

`public static final long` **`DETERMINER`**

Grammatical category of word is determiner. English examples: "the", "a", "some", "many", "his", "her".

CARDINAL

`public static final long` **`CARDINAL`**

Grammatical category of word is cardinal. English examples: "one", "two", "million".

CONJUNCTION

`public static final long` **`CONJUNCTION`**

Grammatical category of word is conjunction. English examples: "and", "or", "since", "if".

PREPOSITION

`public static final long` **`PREPOSITION`**

Grammatical category of word is preposition. English examples: "of", "for".

CONTRACTION

`public static final long` **`CONTRACTION`**

Grammatical category is contraction. English examples: "don’t", "can’t".

ABBREVIATION

`public static final long` **`ABBREVIATION`**

Word is an abbreviation or acronynm. English examples: "Mr.", "USA".

Method Detail

setWrittenForm

`public void` **`setWrittenForm`**`(java.lang.String text)`

Set the "written form" of the `Word`. The written form text should be a string that could be used to present the `Word` visually.

---

getWrittenForm

`public java.lang.String` **`getWrittenForm`**`()`

Get the written form of the `Word`.

---

setSpokenForm

`public void` **`setSpokenForm`**`(java.lang.String text)`

Set the "spoken form" of the `Word`. May be `null`.

The spoken form of a word is useful for mapping the written form to words that are likely to be handled by a speech recognizer or synthesizer. For example, "JavaSoft" to "java soft", "toString" -> "to string", "IEEE" -> "I triple E".

---

getSpokenForm

`public java.lang.String` **`getSpokenForm`**`()`

Get the "spoken form" of the `Word`. Returns `null` if the spoken form is not defined.

---

setPronunciations

`public void` **`setPronunciations`**`(java.lang.String[] pron)`

Set the pronunciation of the `Word` as an array containing a phonetic character `String` for each pronunciation of the word.

The pronunciation string uses the IPA subset of Unicode.

The string should be `null` if no pronunciation is available. Speech engines should be expected to handle most words of the language they support.

Recognizers can use pronunciation information to improve recognition accuracy. Synthesizers use the information to accurately speak unusual words (e.g., foreign words).

---

getPronunciations

`public java.lang.String[] **getPronunciations**()`

Get the pronunciations of the `Word`. The pronunciation string uses the Unicode IPA subset. Returns `null` if no pronunciations are specified.

---

setCategories

`public void **setCategories**(long cat)`

Set the categories of the `Word`. The categories may be `UNKNOWN` or may be an OR'ed set of the defined categories such as `NOUN`, `VERB`, `PREPOSITION`. For example:

```
Word w = new Word("running");
w.setCategories(Word.NOUN | Word.VERB);
```

The category information is a guide to the word's grammatical role. Speech synthesizers can use this information to improve phrasing and accenting.

---

getCategories

`public long **getCategories**()`

Get the categories of the `Word`. Value may be `UNKNOWN` or an OR'ed set of the categories defined by this class.

Package javax.speech.synthesis

Interface Summary	
Speakable	An object implementing the `Speakable` interface can be provided to the `speak` method of a `Synthesizer` to be spoken.
SpeakableListener	The listener interface for receiving notification of events during spoken output of a `Speakable`.
Synthesizer	The `Synthesizer` interface provides primary access to speech synthesis capabilities.
SynthesizerListener	An extension to the `EngineListener` interface for receiving notification of events associated with a `Synthesizer`.
SynthesizerProperties	Provides control of the run-time properties of a `Synthesizer`.

Class Summary	
SpeakableAdapter	Adapter that receives events associated with spoken output of a `Speakable` object.
SpeakableEvent	Event issued during spoken output of text.
SynthesizerAdapter	Adapter that receives events associated with a `Synthesizer`.
SynthesizerEvent	Event issued by `Synthesizer` to indicate a change in state or other activity.
SynthesizerModeDesc	`SynthesizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech synthesizers.
SynthesizerQueueItem	Represents an object on the speech output queue of a `Synthesizer`.
Voice	A description of one output voice of a speech synthesizer.

Exception Summary	
JSMLException	Thrown if a syntax problem is found with text in the marked up with the Java Speech Markup Language.

Class javax.speech.synthesis.JSMLException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
                    |
                    +--javax.speech.synthesis.JSMLException
```

---

public class **JSMLException**
extends SpeechException

Thrown if a syntax problem is found with text in the marked up with the Java Speech Markup Language.

The exception message is a printable string.

See Also:
Serialized Form

Constructor Summary

JSMLException() Constructs a `JSMLException` with no detail message.
JSMLException(java.lang.String s) Constructs a `JSMLException` with the specified detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

JSMLException

`public JSMLException()`

Constructs a `JSMLException` with no detail message.

JSMLException

`public JSMLException(java.lang.String s)`

Constructs a `JSMLException` with the specified detail message.

Parameters:
`s` - a printable detail message

Interface javax.speech.synthesis.Speakable

public abstract interface **Speakable**

An object implementing the `Speakable` interface can be provided to the `speak` method of a `Synthesizer` to be spoken. The text is accessed through the `getJSMLText` method making it the spoken equivalent of the `toString` method of a Java object.

Applications can extend (nearly) any Java object to implement the `Speakable` interface (strictly speaking, any non-final object). Examples might include graphical objects or database entries.

The `getJSMLText` method returns text formatted for the Java Speech Markup Language -- defined in the Java Speech Markup Language specification. JSML allows structural information (paragraphs and sentences), production information (pronunciations, emphasis, breaks, and prosody), and other miscellaneous markup. Appropriate use of this markup improves the quality and understandability of the synthesized speech.

The JSML text is a Unicode string and is assumed to contain text of a single language (the language of the `Synthesizer`). The text is treated as independent of other text output on the synthesizer's text output queue, so, a sentence or other important structure should be contained within a single speakable object.

The standard XML header is optional for software-created JSML documents. Thus, the `getJSMLText` method is not required to provide the header.

A `SpeakableListener` can be attached to the `Synthesizer` with the `addSpeakableListener` method to receive all `SpeakableEvents` for all `Speakable` objects on the output queue.

See Also:
SpeakableListener, SpeakableEvent, Synthesizer, speak(Speakable, SpeakableListener), addSpeakableListener

---

Method Summary

java.lang.String	**getJSMLText**() Return text to be spoken formatted for the Java Speech Markup Language.

Method Detail

getJSMLText

`public java.lang.String **getJSMLText**()`

Return text to be spoken formatted for the Java Speech Markup Language. This method is called immediately when a `Speakable` object is passed to the `speak` method of a `Synthesizer`. The text placed on the speaking queue can be inspected through the `SynthesizerQueueItem` on the speech output queue available through the synthesizer's `enumerateQueue` method.

Returns:
a string containing Java Speech Markup Language text

See Also:
speak(Speakable, SpeakableListener), enumerateQueue, SynthesizerQueueItem

---

Class javax.speech.synthesis.SpeakableAdapter

```
java.lang.Object
  |
  +--javax.speech.synthesis.SpeakableAdapter
```

---

public class **SpeakableAdapter**
extends java.lang.Object
implements SpeakableListener

Adapter that receives events associated with spoken output of a `Speakable` object. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**markerReached**(SpeakableEvent e) A `MARKER_REACHED` event has occurred.
void	**speakableCancelled**(SpeakableEvent e) A `SPEAKABLE_CANCELLED` event has occurred.
void	**speakableEnded**(SpeakableEvent e) A `SPEAKABLE_ENDED` event has occurred.
void	**speakablePaused**(SpeakableEvent e) A `SPEAKABLE_PAUSED` event has occurred.
void	**speakableResumed**(SpeakableEvent e) A `SPEAKABLE_RESUMED` event has occurred.
void	**speakableStarted**(SpeakableEvent e) A `SPEAKABLE_STARTED` event has occurred.
void	**topOfQueue**(SpeakableEvent e) A `TOP_OF_QUEUE` event has occurred.
void	**wordStarted**(SpeakableEvent e) A `WORD_STARTED` event has occurred.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

topOfQueue

`public void topOfQueue(SpeakableEvent e)`

A `TOP_OF_QUEUE` event has occurred.

Specified by:
topOfQueue in interface SpeakableListener
See Also:
TOP_OF_QUEUE

---

speakableStarted

`public void speakableStarted(SpeakableEvent e)`

A `SPEAKABLE_STARTED` event has occurred.

Specified by:
speakableStarted in interface SpeakableListener
See Also:
SPEAKABLE_STARTED

---

speakableEnded

`public void speakableEnded(SpeakableEvent e)`

A `SPEAKABLE_ENDED` event has occurred.

Specified by:
speakableEnded in interface SpeakableListener
See Also:
SPEAKABLE_ENDED

---

speakablePaused

`public void speakablePaused(SpeakableEvent e)`

A `SPEAKABLE_PAUSED` event has occurred.

Specified by:
speakablePaused in interface SpeakableListener
See Also:
SPEAKABLE_PAUSED

---

speakableResumed

`public void speakableResumed(SpeakableEvent e)`

A `SPEAKABLE_RESUMED` event has occurred.

Specified by:
speakableResumed in interface SpeakableListener
See Also:
SPEAKABLE_RESUMED

---

speakableCancelled

`public void **speakableCancelled**(SpeakableEvent e)`

A `SPEAKABLE_CANCELLED` event has occurred.

Specified by:
speakableCancelled in interface SpeakableListener
See Also:
SPEAKABLE_CANCELLED

---

wordStarted

`public void **wordStarted**(SpeakableEvent e)`

A `WORD_STARTED` event has occurred.

Specified by:
wordStarted in interface SpeakableListener
See Also:
WORD_STARTED

---

markerReached

`public void **markerReached**(SpeakableEvent e)`

A `MARKER_REACHED` event has occurred.

Specified by:
markerReached in interface SpeakableListener
See Also:
MARKER_REACHED

Class javax.speech.synthesis.SpeakableEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.synthesis.SpeakableEvent
```

public class **SpeakableEvent**
extends SpeechEvent

Event issued during spoken output of text.

`SpeakableEvents` are issued to `SpeakableListeners`. A single `SpeakableListener` can be provided to any of the `speak` and `speakPlainText` methods of a `Synthesizer` to monitor progress of a single item on the speech output queue. Any number of `SpeakableListener` objects can be attached to a `Synthesizer` with the `addSpeakableListener` method. These listeners receive every `SpeakableEvent` for every item on the speech output queue of the `Synthesizer`. The `SpeakableListener` attached to an individual item on the speech output queue is notified before the `SpeakableListeners` attached to the `Synthesizer`.

The source for a `SpeakableEvent` is the object from which the JSML text was obtained: a `Speakable` object, a `URL`, or a `String`.

The normal sequence of events during output of the item of the top of the synthesizer's speech output is:

- `TOP_OF_QUEUE`
- `SPEAKABLE_STARTED`
- Any number of `WORD_STARTED` and `MARKER_REACHED` events
- `SPEAKABLE_ENDED`

A `SPEAKABLE_PAUSED` may occur any time after the `TOP_OF_QUEUE` but before the `SPEAKABLE_ENDED` event. A `SPEAKABLE_PAUSED` event can only be followed by a `SPEAKABLE_RESUMED` or `SPEAKABLE_CANCELLED`.

A `SPEAKABLE_CANCELLED` event can occur at any time before an `SPEAKABLE_ENDED` (including before a `TOP_OF_QUEUE` event). No other events can follow the `SPEAKABLE_CANCELLED` event since the item has been removed from the speech output queue.

A `SPEAKABLE_CANCELLED` event can be issued for items that are not at the top of the speech output queue. The other events are only issued for the top-of-queue item.

See Also:
SpeakableListener, addSpeakableListener, Speakable, speak(javax.speech.synthesis.Speakable, javax.speech.synthesis.SpeakableListener), speak(java.net.URL, javax.speech.synthesis.SpeakableListener), speak(java.lang.String, javax.speech.synthesis.SpeakableListener), speakPlainText(java.lang.String, javax.speech.synthesis.SpeakableListener), Serialized Form

Field Summary

static int	**ELEMENT_CLOSE** The type of `MARKER_REACHED` event issued at the close of a JSML container element that has a `MARK` attribute on the matching openning tag.
static int	**ELEMENT_EMPTY** The type of `MARKER_REACHED` event issued when an empty JSML element with a `MARK` attribute is reached.
static int	**ELEMENT_OPEN** The type of `MARKER_REACHED` event issued at the openning of a JSML container element with a `MARK` attribute.
static int	**MARKER_REACHED** Issued when audio output reaches a marker contained in the JSML text of a speech output queue item.
int	**markerType** Marker type for a `MARKER_REACHED` event.
static int	**SPEAKABLE_CANCELLED** Issued when an item on the synthesizer's speech output queue is cancelled and removed from the queue.
static int	**SPEAKABLE_ENDED** Issued with the completion of audio output of an object on the speech output queue as the object is removed from the queue.
static int	**SPEAKABLE_PAUSED** Issued when audio output of the item at the top of a synthesizer's speech output queue is paused.
static int	**SPEAKABLE_RESUMED** Issued when audio output of the item at the top of a synthesizer's speech output queue is resumed after a previous pause.
static int	**SPEAKABLE_STARTED** Issued at the start of audio output of an item on the speech output queue.
java.lang.String	**text** The text associated with the `SpeakableEvent`.
static int	**TOP_OF_QUEUE** Issued when an item on the synthesizer's speech output queue reaches the top of the queue.
static int	**WORD_STARTED** Issued when a synthesis engine starts the audio output of a word in the speech output queue item.
int	**wordEnd** Index of last character of of word in JSML text for a `WORD_STARTED` event.
int	**wordStart** Index of first character of of word in JSML text for a `WORD_STARTED` event.

Fields inherited from class javax.speech.SpeechEvent	
id	

Fields inherited from class java.util.EventObject	
source	

Constructor Summary

SpeakableEvent(java.lang.Object source, int id) Constructs an `SpeakableEvent` with a specified source and identifier.
SpeakableEvent(java.lang.Object source, int id, java.lang.String text, int markerType) Constructs an `SpeakableEvent` with a specified source, identifier, text and marker type (used for a `MARKER_REACHED` event).
SpeakableEvent(java.lang.Object source, int id, java.lang.String text, int wordStart, int wordEnd) Constructor for a specified source, identifier, text, wordStart and wordEnd (called for a `WORD_STARTED` event).

Method Summary

int	**getMarkerType**() Return the type of a `MARKER_REACHED` event.
java.lang.String	**getText**() Get the text associated with the event.
int	**getWordEnd**() For a `WORD_STARTED` event, return the index of the last character of the word in the JSML text.
int	**getWordStart**() For a `WORD_STARTED` event, return the index of the first character of the word in the JSML text.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.SpeechEvent	
getId, paramString, toString	

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

TOP_OF_QUEUE

`public static final int **TOP_OF_QUEUE**`

Issued when an item on the synthesizer's speech output queue reaches the top of the queue. If the `Synthesizer` is not paused, the `TOP_OF_QUEUE` event will be followed immediately by the `SPEAKABLE_STARTED` event. If the `Synthesizer` is paused, the `SPEAKABLE_STARTED` event will be delayed until the `Synthesizer` is resumed.

A `QUEUE_UPDATED` is also issued when the speech output queue changes (e.g. a new item at the top of the queue). The `SpeakableEvent` is issued prior to the `SynthesizerEvent`.

See Also:
topOfQueue, QUEUE_UPDATED, PAUSED

SPEAKABLE_STARTED

`public static final int **SPEAKABLE_STARTED**`

Issued at the start of audio output of an item on the speech output queue. This event immediately follows the `TOP_OF_QUEUE` unless the `Synthesizer` is paused when the speakable text is promoted to the top of the output queue.

See Also:
speakableStarted

SPEAKABLE_ENDED

`public static final int **SPEAKABLE_ENDED**`

Issued with the completion of audio output of an object on the speech output queue as the object is removed from the queue.

A `QUEUE_UPDATED` or `QUEUE_EMPTIED` event is also issued when the speech output queue changes because the speech output of the item at the top of queue is completed. The `SpeakableEvent` is issued prior to the `SynthesizerEvent`.

See Also:
speakableEnded, QUEUE_EMPTIED, QUEUE_UPDATED

SPEAKABLE_PAUSED

`public static final int **SPEAKABLE_PAUSED**`

Issued when audio output of the item at the top of a synthesizer's speech output queue is paused. The `SPEAKABLE_PAUSED` `SpeakableEvent` is issued prior to the `ENGINE_PAUSED` event that is issued to the `SynthesizerListener`.

See Also:
speakablePaused, ENGINE_PAUSED

SPEAKABLE_RESUMED

`public static final int **SPEAKABLE_RESUMED**`

Issued when audio output of the item at the top of a synthesizer's speech output queue is resumed after a previous pause. The `SPEAKABLE_RESUMED` `SpeakableEvent` is issued prior to the `ENGINE_RESUMED` event that is issued to the `SynthesizerListener`.

See Also:
speakableResumed, ENGINE_RESUMED

SPEAKABLE_CANCELLED

`public static final int **SPEAKABLE_CANCELLED**`

Issued when an item on the synthesizer's speech output queue is cancelled and removed from the queue. A speech output queue item may be cancelled at any time following a call to `speak`. An item can be cancelled even if it is not at the top of the speech output queue (other `SpeakableEvents` are issued only to the top-of-queue item). Once cancelled, the listener for the cancelled object receives no further `SpeakableEvents`.

The `SPEAKABLE_CANCELLED` `SpeakableEvent` is issued prior to the `QUEUE_UPDATED` or `QUEUE_EMPTIED` event that is issued to the `SynthesizerListener`.

See Also:
speakableCancelled, cancel(), cancel(Object), cancelAll(), QUEUE_UPDATED, QUEUE_EMPTIED

WORD_STARTED

`public static final int WORD_STARTED`

Issued when a synthesis engine starts the audio output of a word in the speech output queue item. The `text`, `wordStart` and `wordEnd` parameters defines the segment of the speakable’s string which is now being spoken.

See Also:
wordStarted, getText, getWordStart, getWordEnd

MARKER_REACHED

`public static final int MARKER_REACHED`

Issued when audio output reaches a marker contained in the JSML text of a speech output queue item. The event text is the string of the `MARK` attribute. The `markerType` indicates whether the mark is at the openning or close of a JSML element or is an attribute of an empty element (no close).

See Also:
markerReached, getText, getMarkerType, ELEMENT_OPEN, ELEMENT_CLOSE, ELEMENT_EMPTY

ELEMENT_OPEN

`public static final int ELEMENT_OPEN`

The type of `MARKER_REACHED` event issued at the openning of a JSML container element with a `MARK` attribute. An `ELEMENT_OPEN` event is followed by an `ELEMENT_CLOSE` event for the closing of the element (unless the `Speakable` is cancelled).

Example: the event for the `MARK` attribute on the openning `SENT` tag will be issued before the start of the word "Testing" in:

```
<SENT MARK="open">Testing one, <MARKER MARK="here"/> two, three.</SENT>
```

See Also:
MARKER_REACHED, getMarkerType

ELEMENT_CLOSE

`public static final int ELEMENT_CLOSE`

The type of `MARKER_REACHED` event issued at the close of a JSML container element that has a `MARK` attribute on the matching openning tag. The `ELEMENT_CLOSE` event always follows a matching `ELEMENT_OPEN` event for the matching openning tag.

Example: the event for the closing `SENT` tag for the `MARK` attribute at the openning of the `SENT` element. The event will be issued after the word "three" is spoken.

```
<SENT MARK="open">Testing one, <MARKER MARK="here"/> two, three.</SENT>
```

See Also:
MARKER_REACHED, getMarkerType

ELEMENT_EMPTY

`public static final int ELEMENT_EMPTY`

The type of `MARKER_REACHED` event issued when an empty JSML element with a `MARK` attribute is reached. (An empty JSML has no closing tag and is indicated by a slash (’/’) before the ’>’ character.)

Example: the `MARKER` tag below is empty event so an `ELEMENT_EMPTY` type of `MARKER_REACHED` event is issued before the word "two" is spoken in:

```
<SENT MARK="open">Testing one, <MARKER MARK="here"/> two, three.</SENT>
```

See Also:
MARKER_REACHED, getMarkerType

text

`protected java.lang.String text`

The text associated with the `SpeakableEvent`.

markerType

`protected int markerType`

Marker type for a `MARKER_REACHED` event.

See Also:
getMarkerType, ELEMENT_OPEN, ELEMENT_CLOSE, ELEMENT_EMPTY

wordStart

`protected int wordStart`

Index of first character of of word in JSML text for a `WORD_STARTED` event.

See Also:
WORD_STARTED, getWordStart

wordEnd

```
protected int wordEnd
```

Index of last character of of word in JSML text for a `WORD_STARTED` event.

See Also:
WORD_STARTED, getWordEnd

Constructor Detail

SpeakableEvent

```
public SpeakableEvent(java.lang.Object source,
                      int id)
```

Constructs an `SpeakableEvent` with a specified source and identifier.

SpeakableEvent

```
public SpeakableEvent(java.lang.Object source,
                      int id,
                      java.lang.String text,
                      int markerType)
```

Constructs an `SpeakableEvent` with a specified source, identifier, text and marker type (used for a `MARKER_REACHED` event).

SpeakableEvent

```
public SpeakableEvent(java.lang.Object source,
                      int id,
                      java.lang.String text,
                      int wordStart,
                      int wordEnd)
```

Constructor for a specified source, identifier, text, wordStart and wordEnd (called for a `WORD_STARTED` event).

Method Detail

getMarkerType

```
public int getMarkerType()
```

Return the type of a `MARKER_REACHED` event.

See Also:
MARKER_REACHED, ELEMENT_OPEN, ELEMENT_CLOSE, ELEMENT_EMPTY

---

getText

```
public java.lang.String getText()
```

Get the text associated with the event.

For `WORD_STARTED`, the text is the next word to be spoken. This text may differ from the text between the `wordStart` and `wordEnd` points is the original JSML text.

For `MARKER_REACHED`, the text is the `MARK` attribute in the JSML text.

See Also:
WORD_STARTED, MARKER_REACHED

---

getWordStart

```
public int getWordStart()
```

For a `WORD_STARTED` event, return the index of the first character of the word in the JSML text.

---

getWordEnd

```
public int getWordEnd()
```

For a `WORD_STARTED` event, return the index of the last character of the word in the JSML text.

---

paramString

```
public java.lang.String paramString()
```

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class SpeechEvent

Interface javax.speech.synthesis.SpeakableListener

Implementing Classes:
SpeakableAdapter

---

public abstract interface **SpeakableListener**
extends java.util.EventListener

The listener interface for receiving notification of events during spoken output of a `Speakable`. Events are requested by either:

- Providing a `SpeakableListener` object when calling one of the `speak` or `speakPlainText` methods of a `Synthesizer`.
- Attaching a `SpeakableListener` to a `Synthesizer` with its `addSpeakableListener` method.

The speakable events and the sequencing of events is defined in the documentation for `SpeakableEvent`. The source of each `SpeakableEvent` is the object from which JSML text was derived: a `Speakable` object, a `URL`, or a `String`.

The `SpeakableAdapter` class provides a trivial implementation of this interface.

See Also:
Speakable, SpeakableEvent, Synthesizer, addSpeakableListener, speak(javax.speech.synthesis.Speakable, javax.speech.synthesis.SpeakableListener), speak(java.net.URL, javax.speech.synthesis.SpeakableListener), speak(java.lang.String, javax.speech.synthesis.SpeakableListener), speakPlainText(java.lang.String, javax.speech.synthesis.SpeakableListener)

Method Summary	
void	**markerReached**(SpeakableEvent e) A `MARKER_REACHED` event has occurred.
void	**speakableCancelled**(SpeakableEvent e) A `SPEAKABLE_CANCELLED` event has occurred.
void	**speakableEnded**(SpeakableEvent e) A `SPEAKABLE_ENDED` event has occurred.
void	**speakablePaused**(SpeakableEvent e) A `SPEAKABLE_PAUSED` event has occurred.
void	**speakableResumed**(SpeakableEvent e) A `SPEAKABLE_RESUMED` event has occurred.
void	**speakableStarted**(SpeakableEvent e) A `SPEAKABLE_STARTED` event has occurred.
void	**topOfQueue**(SpeakableEvent e) A `TOP_OF_QUEUE` event has occurred.
void	**wordStarted**(SpeakableEvent e) A `WORD_STARTED` event has occurred.

Method Detail

topOfQueue

`public void topOfQueue(SpeakableEvent e)`

A `TOP_OF_QUEUE` event has occurred.

See Also:
TOP_OF_QUEUE

speakableStarted

`public void speakableStarted(SpeakableEvent e)`

A `SPEAKABLE_STARTED` event has occurred.

See Also:
SPEAKABLE_STARTED

speakableEnded

`public void speakableEnded(SpeakableEvent e)`

A `SPEAKABLE_ENDED` event has occurred.

See Also:
SPEAKABLE_ENDED

---

speakablePaused

`public void speakablePaused(SpeakableEvent e)`

A `SPEAKABLE_PAUSED` event has occurred.

See Also:
SPEAKABLE_PAUSED

---

speakableResumed

`public void speakableResumed(SpeakableEvent e)`

A `SPEAKABLE_RESUMED` event has occurred.

See Also:
SPEAKABLE_RESUMED

---

speakableCancelled

`public void speakableCancelled(SpeakableEvent e)`

A `SPEAKABLE_CANCELLED` event has occurred.

See Also:
SPEAKABLE_CANCELLED

---

wordStarted

`public void wordStarted(SpeakableEvent e)`

A `WORD_STARTED` event has occurred.

See Also:
WORD_STARTED

markerReached

`public void markerReached(SpeakableEvent e)`

A `MARKER_REACHED` event has occurred.

See Also:
MARKER_REACHED

Interface javax.speech.synthesis.Synthesizer

public abstract interface **Synthesizer**
extends Engine

The `Synthesizer` interface provides primary access to speech synthesis capabilities. The `Synthesizer` interface extends the `Engine` interface. Thus, any `Synthesizer` implements basic speech engine capabilities plus the specialized capabilities required for speech synthesis.

The primary functions provided by the `Synthesizer` interface are the ability to speak text, speak Java Speech Markup Language text, and control an output queue of objects to be spoken.

Creating a Synthesizer

Typically, a Synthesizer is created by a call to the `Central.createSynthesizer` method. The procedures for locating, selecting, creating and initializing a `Synthesizer` are described in the documentation for `Central` class.

Synthesis Package: Inherited and Extended Capabilities

A `synthesis` package inherits many of its important capabilities from the `Engine` interface and its related support classes and interfaces. The `synthesis` package adds specialized functionality for performing speech synthesis.

- Inherits location mechanism by `Central.availableSynthesizers` method and `EngineModeDesc`.
- Extends `EngineModeDesc` as `SynthesizerModeDesc`.
- Inherits `allocate` and `deallocate` methods from the `Engine` interface.
- Inherits `pause` and `resume` methods from the `Engine` interface.
- Inherits `getEngineState`, `waitEngineState` and `testEngineState` methods from the `Engine` interface.
- Inherits the `DEALLOCATED`, `ALLOCATED`, `ALLOCATING_RESOURCES`, `DEALLOCATING_RESOURCES`, `PAUSED` and `RESUMED` states from the `Engine` interface.
- Adds `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` sub-states to the `ALLOCATED` state.
- Inherits audio management: see `Engine.getAudioManager` and `AudioManager`.
- Inherits vocabulary management: see Engine.getVocabManager and VocabManager.
- Inherits `addEngineListener` and `removeEngineListener` methods and uses the `EngineListener` interface.
- `Extends EngineListener` interface to `SynthesizerListener`.
- Adds `speak(Speakable, Listener)`, `speak(URL, Listener)`, `speak(String, Listener)` and `speakPlainText(String)` methods to place text on the output queue of the synthesizer.
- Adds `phoneme(String)` method that converts text to phonemes.
- Adds `enumerateQueue`, `cancel()`, `cancel(Object)` and `cancelAll` methods for management of output queue.

Speaking Text

The basic function of a `Synthesizer` is to speak text provided to it by an application. This text can be plain Unicode text in a `String` or can be marked up using the Java Speech Markup Language (JSML).

Plain text is spoken using the `speakPlainText` method. JSML text is spoken using one of the three `speak` methods. The `speak` methods obtain the JSML text for a `Speakable` object, from a `URL`, or from a `String`.

[Note: JSML text provided programmatically (by a `Speakable` object or a `String`) does not require the full XML header. JSML text obtained from a URL requires the full XML header.]

A synthesizer is mono-lingual (it speaks a single language) so the text should contain only the single language of the synthesizer. An application requiring output of more than one language needs to create multiple `Synthesizer` object through `Central`. The language of the `Synthesizer` should be selected at the time at which it is created. The language for a created `Synthesizer` can be checked through the `Locale` of its `EngineModeDesc` (see `getEngineModeDesc`).

Each object provided to a synthesizer is spoken independently. Sentences, phrases and other structures should not span multiple call to the `speak` methods.

Synthesizer State System

`Synthesizer` extends the state system of the generic `Engine` interface. It inherits the four basic allocation states, plus the `PAUSED` and `RESUMED` states.

`Synthesizer` adds a pair of sub-states to the `ALLOCATED` state to represent the state of the speech output queue (queuing is described in more detail below). For an `ALLOCATED Synthesizer`, the speech output queue is either empty or not empty: represented by the states `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY`.

The queue status is independent of the pause/resume status. Pausing or resuming a synthesizer does not effect the queue. Adding or removing objects from the queue does not effect the pause/resume status. The only form of interaction between these state systems is that the `Synthesizer` only speaks in the `RESUMED` state, and therefore, a transition from `QUEUE_NOT_EMPTY` to `QUEUE_EMPTY` because of completion of speaking an object is only possible in the `RESUMED` state. (A transition from `QUEUE_NOT_EMPTY` to `QUEUE_EMPTY` is possible in the `PAUSED` state only through a call to one of the `cancel` methods.)

Speech Output Queue

A synthesizer implements a queue of items provided to it through the `speak` and `speakPlainText` methods. The queue is "first-in, first-out (FIFO)" -- the objects are spoken in exactly he order in which they are received. The object at the top of the queue is the object that is currently being spoken or about to be spoken.

The `QUEUE_EMPTY` and `QUEUE_NOT_EMPTY` states of a `Synthesizer` indicate the current state of of the speech output queue. The state handling methods inherited from the `Engine` interface (`getEngineState`, `waitEngineState` and `testEngineState`) can be used to test the queue state.

The items on the queue can be checked with the `enumerateQueue` method which returns a snapshot of the queue.

The `cancel` methods allows an application to (a) stop the output of item currently at the top of the speaking queue, (b) remove an arbitrary item from the queue, or (c) remove all items from the output queue.

Applications requiring more complex queuing mechanisms (e.g. a prioritized queue) can implement their own queuing objects that control the synthesizer.

Pause and Resume

The pause and resume methods (inherited from the `javax.speech.Engine` interface) have behavior like a "tape player". Pause stops audio output as soon as possible. Resume restarts audio output from the point of the pause. Pause and resume may occur within words, phrases or unnatural points in the speech output.

Pause and resume do not affect the speech output queue.

In addition to the `ENGINE_PAUSED` and `ENGINE_RESUMED` events issued to the `EngineListener` (or `SynthesizerListener`), `SPEAKABLE_PAUSED` and `SPEAKABLE_RESUMED` events are issued to appropriate `SpeakableListeners` for the `Speakable` object at the top of the speaking queue. (The `SpeakableEvent` is first issued to any `SpeakableListener` provided with the `speak` method, then to each `SpeakableListener` attached to the `Synthesizer`. Finally, the `EngineEvent` is issued to each `SynthesizerListener` and `EngineListener` attached to the `Synthesizer`.)

Applications can determine the approximate point at which a pause occurs by monitoring the `WORD_STARTED` events.

See Also:
Central, Speakable, SpeakableListener, EngineListener, SynthesizerListener

---

Field Summary

static long	**QUEUE_EMPTY** Bit of state that is set when the speech output queue of a `Synthesizer` is empty.
static long	**QUEUE_NOT_EMPTY** Bit of state that is set when the speech output queue of a `Synthesizer` is not empty.

Method Summary

void	**addSpeakableListener**(SpeakableListener listener) Request notifications of all `SpeakableEvents` for all speech output objects for this `Synthesizer`.
void	**cancelAll**() Cancel all objects in the synthesizer speech output queue and stop speaking the current top-of-queue object.
void	**cancel**() Cancel output of the current object at the top of the output queue.
void	**cancel**(java.lang.Object source) Remove a specified item from the speech output queue.
java.util.Enumeration	**enumerateQueue**() Return an `Enumeration` containing a snapshot of all the objects currently on the speech output queue.
SynthesizerProperties	**getSynthesizerProperties**() Return the `SynthesizerProperties` object (a JavaBean).
java.lang.String	**phoneme**(java.lang.String text) Returns the phoneme string for a text string.
void	**removeSpeakableListener**(SpeakableListener listener) Remove a `SpeakableListener` from this `Synthesizer`.
void	**speakPlainText**(java.lang.String text, SpeakableListener listener) Speak a plain text string.
void	**speak**(Speakable JSMLtext, SpeakableListener listener) Speak an object that implements the `Speakable` interface and provides text marked with the Java Speech Markup Language.
void	**speak**(java.net.URL JSMLurl, SpeakableListener listener) Speak text from a URL formatted with the Java Speech Markup Language.
void	**speak**(java.lang.String JSMLText, SpeakableListener listener) Speak a string containing text formatted with the Java Speech Markup Language.

Field Detail

QUEUE_EMPTY

`public static final long` **`QUEUE_EMPTY`**

Bit of state that is set when the speech output queue of a `Synthesizer` is empty. The `QUEUE_EMPTY` state is a sub-state of the `ALLOCATED` state. An allocated `Synthesizer` is always in either the `QUEUE_NOT_EMPTY` or `QUEUE_EMPTY` state.

A `Synthesizer` is always allocated in the `QUEUE_EMPTY` state. The `Synthesizer` transitions from the `QUEUE_EMPTY` state to the `QUEUE_NOT_EMPTY` state when a call to one of the `speak` methods places an object on the speech output queue. A `QUEUE_UPDATED` event is issued to indicate this change in state.

A `Synthesizer` returns from the `QUEUE_NOT_EMPTY` state to the `QUEUE_EMPTY` state once the queue is emptied because of completion of speaking all objects or because of a `cancel`.

The queue status can be tested with the `waitQueueEmpty`, `getEngineState` and `testEngineState` methods. To block a thread until the queue is empty:

```
Synthesizer synth = ...;
synth.waitEngineState(QUEUE_EMPTY);
```

See Also:
QUEUE_NOT_EMPTY, ALLOCATED, getEngineState, waitEngineState, testEngineState, QUEUE_UPDATED

QUEUE_NOT_EMPTY

```
public static final long QUEUE_NOT_EMPTY
```

Bit of state that is set when the speech output queue of a `Synthesizer` is not empty. The `QUEUE_NOT_EMPTY` state is a sub-state of the `ALLOCATED` state. An allocated `Synthesizer` is always in either the `QUEUE_NOT_EMPTY` or `QUEUE_EMPTY` state.

A `Synthesizer` enters the `QUEUE_NOT_EMPTY` from the `QUEUE_EMPTY` state when one of the `speak` methods is called to place an object on the speech output queue. A `QUEUE_UPDATED` event is issued to mark this change in state.

A `Synthesizer` returns from the `QUEUE_NOT_EMPTY` state to the `QUEUE_EMPTY` state once the queue is emptied because of completion of speaking all objects or because of a `cancel`.

See Also:
QUEUE_EMPTY, ALLOCATED, getEngineState, waitEngineState, testEngineState, QUEUE_UPDATED

Method Detail

speak

```
public void speak(Speakable JSMLtext,
                  SpeakableListener listener)
           throws JSMLException,
                  EngineStateError
```

Speak an object that implements the `Speakable` interface and provides text marked with the Java Speech Markup Language. The `Speakable` object is added to the end of the speaking queue and will be spoken once it reaches the top of the queue and the synthesizer is in the `RESUMED` state.

The synthesizer first requests the text of the `Speakable` by calling its `getJSMLText` method. It then checks the syntax of the JSML markup and throws a `JSMLException` if any problems are found. If the JSML text is legal, the text is placed on the speech output queue.

When the speech output queue is updated, a `QUEUE_UPDATE` event is issued to `SynthesizerListeners`.

Events associated with the `Speakable` object are issued to the `SpeakableListener` object. The listener may be `null`. A listener attached with this method cannot be removed with a subsequent remove call. The source for the `SpeakableEvents` is the `JSMLtext` object.

`SpeakableEvents` can also be received by attaching a `SpeakableListener` to the `Synthesizer` with the `addSpeakableListener` method. A `SpeakableListener` attached to the `Synthesizer` receives all `SpeakableEvents` for all speech output items of the synthesizer (rather than for a single `Speakable`).

The speak call is asynchronous: it returns once the text for the `Speakable` has been obtained, checked for syntax, and placed on the synthesizer's speech output queue. An application needing to know when the `Speakable` has been spoken should wait for the `SPEAKABLE_ENDED` event to be issued to the `SpeakableListener` object. The `getEngineState`, `waitEngineState` and `enumerateQueue` methods can be used to determine the speech output queue status.

An object placed on the speech output queue can be removed with one of the `cancel` methods.

The `speak` methods operate as defined only when a `Synthesizer` is in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
`JSMLText` - object implementing the Speakable interface that provides Java Speech Markup Language text to be spoken
`listener` - receives notification of events as synthesis output proceeds
Throws:
JSMLException - if any syntax errors are encountered in `JSMLtext`
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
speak(String, SpeakableListener), speak(URL, SpeakableListener), speakPlainText(String, SpeakableListener), SpeakableEvent, addSpeakableListener

---

speak

```
public void speak(java.net.URL JSMLurl,
                  SpeakableListener listener)
           throws JSMLException,
                  java.net.MalformedURLException,
                  java.io.IOException,
                  EngineStateError
```

Speak text from a URL formatted with the Java Speech Markup Language. The text is obtained from the URL, checked for legal JSML formatting, and placed at the end of the speaking queue. It is spoken once it reaches the top of the queue and the synthesizer is in the `RESUMED` state. In other respects is it identical to the `speak` method that accepts a `Speakable` object.

The source of a `SpeakableEvent` issued to the `SpeakableListener` is the `URL`.

Because of the need to check JSML syntax, this `speak` method returns only once the complete URL is loaded, or until a syntax error is detected in the URL stream. Network delays will cause the method to return slowly.

Note: the full XML header is required in the JSML text provided in the URL. The header is optional on programmatically generated JSML (ie. with the `speak(String, Listener)` and `speak(Speakable, Listener)` methods.

The `speak` methods operate as defined only when a `Synthesizer` is in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
- `JSMLurl` - URL containing Java Speech Markup Language text to be spoken
- `JSMLException` - if any syntax errors are encountered in `JSMLtext`
- `listener` - receives notification of events as synthesis output proceeds

Throws:
- EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
- speak(Speakable, SpeakableListener), speak(String, SpeakableListener), speakPlainText(String, SpeakableListener), SpeakableEvent, addSpeakableListener

---

speak

```
public void speak(java.lang.String JSMLText,
                  SpeakableListener listener)
        throws JSMLException,
               EngineStateError
```

Speak a string containing text formatted with the Java Speech Markup Language. The JSML text is checked for formatting errors and a `JSMLException` is thrown if any are found. If legal, the text is placed at the end of the speaking queue and will be spoken once it reaches the top of the queue and the synthesizer is in the `RESUMED` state. In all other respects is it identical to the `speak` method that accepts a `Speakable` object.

The source of a `SpeakableEvent` issued to the `SpeakableListener` is the `String`.

The `speak` methods operate as defined only when a `Synthesizer` is in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
- `JSMLText` - String contains Java Speech Markup Language text to be spoken
- `listener` - receives notification of events as synthesis output proceeds
- `JSMLException` - if any syntax errors are encountered in `JSMLtext`

Throws:
- EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
- speak(Speakable, SpeakableListener), speak(URL, SpeakableListener), speakPlainText(String, SpeakableListener)

---

speakPlainText

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

Speak a plain text string. The text is not interpreted as containing the Java Speech Markup Language so JSML elements are ignored. The text is placed at the end of the speaking queue and will be spoken once it reaches the top of the queue and the synthesizer is in the `RESUMED` state. In other respects it is similar to the `speak` method that accepts a `Speakable` object.

The source of a `SpeakableEvent` issued to the `SpeakableListener` is the `String` object.

The `speak` methods operate as defined only when a `Synthesizer` is in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
- `JSMLText` - String contains plaing text to be spoken
- `listener` - receives notification of events as synthesis output proceeds

Throws:
- EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
- speak(Speakable, SpeakableListener), speak(URL, SpeakableListener), speak(String, SpeakableListener)

---

phoneme

```
public java.lang.String phoneme(java.lang.String text)
                         throws EngineStateError
```

Returns the phoneme string for a text string. The return string uses the International Phonetic Alphabet subset of Unicode. The input string is expected to be simple text (for example, a word or phrase in English). The text is not expected to contain punctuation or JSML markup.

If the `Synthesizer` does not support text-to-phoneme conversion or cannot process the input text it will return `null`.

If the text has multiple pronunciations, there is no way to indicate which pronunciation is preferred.

The `phoneme` method operate as defined only when a `Synthesizer` is in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
`text` - plain text to be converted to phonemes
Returns:
phonemic representation of text or `null`
Throws:
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

enumerateQueue

```
public java.util.Enumeration enumerateQueue()
                                     throws EngineStateError
```

Return an `Enumeration` containing a snapshot of all the objects currently on the speech output queue. The first item is the top of the queue. An empty queue returns a null object.

If the return value is non-null then each object it contains is guaranteed to be a `SynthesizerQueueItem` object representing the source object (`Speakable` object, `URL`, or a `String`) and the JSML or plain text obtained from that object.

A `QUEUE_UPDATED` event is issued to each `SynthesizerListener` whenever the speech output queue changes. A `QUEUE_EMPTIED` event is issued whenever the queue the emptied.

This method returns only the items on the speech queue placed there by the current application or applet. For security reasons, it is not possible to inspect items placed by other applications.

The items on the speech queue cannot be modified by changing the object returned from this method.

The `enumerateQueue` method works in the `ALLOCATED` state. The call blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Returns:
an `Enumeration` of the speech output queue or null
Throws:
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
SynthesizerQueueItem, QUEUE_UPDATED, QUEUE_EMPTIED, addEngineListener

---

cancel

```
public void cancel()
            throws EngineStateError
```

Cancel output of the current object at the top of the output queue. A `SPEAKABLE_CANCELLED` event is issued to appropriate `SpeakableListeners`.

If there is another object in the speaking queue, it is moved to top of queue and receives the `TOP_OF_QUEUE` event. If the `Synthesizer` is not paused, speech output continues with that object. To prevent speech output continuing with the next object in the queue, call `pause` before calling `cancel`.

A `SynthesizerEvent` is issued to indicate `QUEUE_UPDATED` (if objects remain on the queue) or `QUEUE_EMPTIED` (if the cancel leaves the queue empty).

It is not an exception to call cancel if the speech output queue is empty.

The `cancel` methods work in the `ALLOCATED` state. The calls blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and complete when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Throws:
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
cancel(Object), cancelAll(), QUEUE_UPDATED, QUEUE_EMPTIED, TOP_OF_QUEUE, SPEAKABLE_CANCELLED

---

cancel

```
public void cancel(java.lang.Object source)
            throws java.lang.IllegalArgumentException,
                   EngineStateError
```

Remove a specified item from the speech output queue. The source object must be one of the items passed to a `speak` method. A `SPEAKABLE_CANCELLED` event is issued to appropriate `SpeakableListeners`.

If the source object is the top item in the queue, the behavior is the same as the `cancel()` method.

If the source object is not at the top of the queue, it is removed from the queue without affecting the current top-of-queue speech output. A `QUEUE_UPDATED` is then issued to `SynthesizerListeners`.

If the source object appears multiple times in the queue, only the first instance is cancelled.

Warning: cancelling an object just after the synthesizer has completed speaking it and has removed the object from the queue will cause an exception. In this instance, the exception can be ignored.

The `cancel` methods work in the `ALLOCATED` state. The calls blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and complete when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Parameters:
`source` - object to be removed from the speech output queue
Throws:
java.lang.IllegalArgumentException - if the source object is not found in the speech output queue.
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
cancel(), cancelAll(), QUEUE_UPDATED, QUEUE_EMPTIED, SPEAKABLE_CANCELLED

---

cancelAll

```
public void cancelAll()
               throws EngineStateError
```

Cancel all objects in the synthesizer speech output queue and stop speaking the current top-of-queue object.

The `SpeakableListeners` of each cancelled item on the queue receive a `SPEAKABLE_CANCELLED` event. A `QUEUE_EMPTIED` event is issued to attached `SynthesizerListeners`.

A `cancelAll` is implictly performed before a `Synthesizer` is deallocated.

The `cancel` methods work in the `ALLOCATED` state. The calls blocks if the `Synthesizer` in the `ALLOCATING_RESOURCES` state and complete when the engine reaches the `ALLOCATED` state. An error is thrown for synthesizers in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states.

Throws:
EngineStateError - if called for a synthesizer in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
cancel(), cancel(Object), QUEUE_EMPTIED, SPEAKABLE_CANCELLED

---

getSynthesizerProperties

```
public SynthesizerProperties getSynthesizerProperties()
```

Return the `SynthesizerProperties` object (a JavaBean). The method returns exactly the same object as the `getEngineProperties` method in the `Engine` interface. However, with the `getSynthesizerProperties` method, an application does not need to cast the return value.

The `SynthesizerProperties` are available in any state of an `Engine`. However, changes only take effect once an engine reaches the `ALLOCATED` state.

Returns:
the `SynthesizerProperties` object for this engine
See Also:
getEngineProperties

---

addSpeakableListener

```
public void addSpeakableListener(SpeakableListener listener)
```

Request notifications of all `SpeakableEvents` for all speech output objects for this `Synthesizer`. An application can attach multiple `SpeakableListeners` to a `Synthesizer`. A single listener can be attached to multiple synthesizers.

When an event effects more than one item in the speech output queue (e.g. `cancelAll`), the `SpeakableEvents` are issued in the order of the items in the queue starting with the top of the queue.

A `SpeakableListener` can also provided for an indivudal speech output item by providing it as a parameter to one of the `speak` or `speakPlainText` methods.

A `SpeakableListener` can be attached or removed in any `Engine` state.

Parameters:
`listener` - the listener that will receive `SpeakableEvents`
See Also:
removeSpeakableListener

---

removeSpeakableListener

```
public void removeSpeakableListener(SpeakableListener listener)
```

Remove a `SpeakableListener` from this `Synthesizer`.

A `SpeakableListener` can be attached or removed in any `Engine` state.

See Also:
addSpeakableListener

Class javax.speech.synthesis.SynthesizerAdapter

```
java.lang.Object
  |
  +--javax.speech.EngineAdapter
        |
        +--javax.speech.synthesis.SynthesizerAdapter
```

---

public class **SynthesizerAdapter**
extends EngineAdapter
implements SynthesizerListener

Adapter that receives events associated with a `Synthesizer`. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**queueEmptied**(SynthesizerEvent e) The synthesizer text output queue has emptied.
void	**queueUpdated**(SynthesizerEvent e) The speaking queue has changed.

Methods inherited from class javax.speech.EngineAdapter
engineAllocated, engineAllocatingResources, engineDeallocated, engineDeallocatingResources, engineError, enginePaused, engineResumed

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

queueEmptied

`public void queueEmptied(SynthesizerEvent e)`

The synthesizer text output queue has emptied.

Specified by:
queueEmptied in interface SynthesizerListener

See Also:
QUEUE_EMPTIED

---

queueUpdated

`public void queueUpdated(SynthesizerEvent e)`

The speaking queue has changed.

Specified by:
queueUpdated in interface SynthesizerListener
See Also:
QUEUE_UPDATED

Class javax.speech.synthesis.SynthesizerEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.EngineEvent
                    |
                    +--javax.speech.synthesis.SynthesizerEvent
```

---

public class **SynthesizerEvent**
extends EngineEvent

Event issued by `Synthesizer` to indicate a change in state or other activity. A `SynthesizerEvent` is issued to each `SynthesizerListener` attached to a `Synthesizer` using the `addEngineListener` method it inherits from the `Engine` interface.

The `SynthesizerEvent` class extends the `EngineEvent` class. Similarly, the `SynthesizerListener` interface extends the `EngineListener` interface.

`SynthesizerEvent` extends `EngineEvent` with several events that are specialized for speech synthesis. It also inherits several event types from `EngineEvent`: `ENGINE_ALLOCATED`, `ENGINE_DEALLOCATED`, `ENGINE_ALLOCATING_RESOURCES`, `ENGINE_DEALLOCATING_RESOURCES`, `ENGINE_PAUSED`, `ENGINE_RESUMED`.

See Also:
Synthesizer, SynthesizerListener, addEngineListener, EngineEvent, Serialized Form

---

Field Summary

static int	**QUEUE_EMPTIED** The speaking queue of the `Synthesizer` has emptied and the `Synthesizer` has changed to the `QUEUE_EMPTY` state.
static int	**QUEUE_UPDATED** The speech output queue has changed.
boolean	**topOfQueueChanged** `topOfQueueChanged` is `true` for `QUEUE_UPDATED` event when the top item in the speech output queue has changed.

Fields inherited from class javax.speech.EngineEvent
ENGINE_ALLOCATED, ENGINE_ALLOCATING_RESOURCES, ENGINE_DEALLOCATED, ENGINE_DEALLOCATING_RESOURCES, ENGINE_PAUSED, ENGINE_RESUMED, newEngineState, oldEngineState

Fields inherited from class javax.speech.SpeechEvent
id

Fields inherited from class java.util.EventObject
source

Constructor Summary

SynthesizerEvent(Synthesizer source, int id, boolean topOfQueueChanged, long oldEngineState, long newEngineState) Construct a `SynthesizerEvent` with a specified event id and `topOfQueueChanged` flag.

Method Summary

boolean	**getTopOfQueueChanged**() Return the `topOfQueueChanged` value.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.EngineEvent
getNewEngineState, getOldEngineState, paramString

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

QUEUE_EMPTIED

```
public static final int QUEUE_EMPTIED
```

The speaking queue of the `Synthesizer` has emptied and the `Synthesizer` has changed to the `QUEUE_EMPTY` state. The queue may become empty because speech output of all items in the queue is completed, or because the items have been cancelled.

The `QUEUE_EMPTIED` event follows the `SPEAKABLE_ENDED` or `SPEAKABLE_CANCELLED` event that removed the last item from the speaking queue.

See Also:
queueEmptied, cancel(), cancel(Object), cancelAll(), SPEAKABLE_ENDED, SPEAKABLE_CANCELLED, QUEUE_EMPTY

QUEUE_UPDATED

```
public static final int QUEUE_UPDATED
```

The speech output queue has changed. This event may indicate a change in state of the `Synthesizer` from `QUEUE_EMPTY` to `QUEUE_NOT_EMPTY`. The event may also occur in the `QUEUE_NOT_EMPTY` state without changing state. the `enumerateQueue` method of `Synthesizer` will return a changed list.

The speech output queue changes when (a) a new item is placed on the queue with a call to one of the `speak` methods, (b) when an item is removed from the queue with one of the cancel methods (without emptying the queue), or (c) when output of the top item of the queue is completed (again, without leaving an empty queue).

The `topOfQueueChanged` boolean parameter is set to true if the top item on the queue has changed.

See Also:
queueUpdated, getTopOfQueueChanged, SPEAKABLE_ENDED, QUEUE_NOT_EMPTY

---

topOfQueueChanged

```
protected boolean topOfQueueChanged
```

`topOfQueueChanged` is `true` for `QUEUE_UPDATED` event when the top item in the speech output queue has changed.

See Also:
getTopOfQueueChanged

Constructor Detail

SynthesizerEvent

```
public SynthesizerEvent(Synthesizer source,
                        int id,
                        boolean topOfQueueChanged,
                        long oldEngineState,
                        long newEngineState)
```

Construct a `SynthesizerEvent` with a specified event id and `topOfQueueChanged` flag.

Parameters:
`source` - the `Synthesizer` that issued the event
`id` - the identifier for the event type
`topOfQueueChanged` - `true` if top item on speech output queue changed
`oldEngineState` - engine state prior to this event
`newEngineState` - engine state following this event

Method Detail

getTopOfQueueChanged

```
public boolean getTopOfQueueChanged()
```

Return the `topOfQueueChanged` value. The value is `true` for a `QUEUE_UPDATED` event when the top item in the speech output queue has changed.

See Also:
topOfQueueChanged

---

paramString

```
public java.lang.String paramString()
```

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class EngineEvent

Interface javax.speech.synthesis.SynthesizerListener

Implementing Classes:
SynthesizerAdapter

---

public abstract interface **SynthesizerListener**
extends EngineListener

An extension to the `EngineListener` interface for receiving notification of events associated with a `Synthesizer`. `SynthesizerListener` objects are attached to and removed from a `Synthesizer` by calling the `addEngineListener` and `removeEngineListener` methods (which `Synthesizer` inherits from the `Engine` interface).

The source for all `SynthesizerEvents` provided to a `SynthesizerListener` is the `Synthesizer`.

The `SynthesizerAdapter` class provides a trivial implementation of this interface.

See Also:
SynthesizerListener, Synthesizer, Engine, addEngineListener, removeEngineListener

---

Method Summary

void	**queueEmptied**(SynthesizerEvent e) An `QUEUE_EMPTIED` event has occurred indicating that the text output queue of the `Synthesizer` has emptied.
void	**queueUpdated**(SynthesizerEvent e) An `QUEUE_UPDATED` event has occurred indicating that the speaking queue has changed.

Method Detail

queueEmptied

`public void queueEmptied(SynthesizerEvent e)`

An `QUEUE_EMPTIED` event has occurred indicating that the text output queue of the `Synthesizer` has emptied. The `Synthesizer` is in the `QUEUE_EMPTY` state.

See Also:
QUEUE_EMPTIED, QUEUE_EMPTY

---

queueUpdated

`public void queueUpdated(SynthesizerEvent e)`

An `QUEUE_UPDATED` event has occurred indicating that the speaking queue has changed. The `Synthesizer` is in the `QUEUE_NOT_EMPTY` state.

See Also:
QUEUE_UPDATED, QUEUE_NOT_EMPTY

Class javax.speech.synthesis.SynthesizerModeDesc

```
java.lang.Object
  |
  +--javax.speech.EngineModeDesc
        |
        +--javax.speech.synthesis.SynthesizerModeDesc
```

---

public class **SynthesizerModeDesc**
extends EngineModeDesc

`SynthesizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech synthesizers. A `SynthesizerModeDesc` inherits engine name, mode name, locale and running properties from `EngineModeDesc`. `SynthesizerModeDesc` adds two properties:

- List of voices provided by the synthesizer
- Voice to be loaded when the synthesizer is started (not used in selection)

Like `EngineModeDesc`, there are two types of `SynthesizerModeDesc`: those created by an application which are used in engine selection, and those created by an engine which descibe a particular mode of operation of the engine. Descriptor provided engines are obtained through the `availableSynthesizers` method of the `Central` class and must have all their features defined. A descriptor created by an application may make any or all of the features `null` which means "don't care" (null features are ignored in engine selection).

Applications can modify application-created descriptors in any way. Applications should never modify a `SynthesizerModeDesc` provided by an engine (i.e. returned by the `availableSynthesizers` method.

Engine creation is described in the documentation for the `Central` class.

See Also:
Central, createSynthesizer, Voice

Constructor Summary

SynthesizerModeDesc() Construct a descriptor with all features set to `null`.
SynthesizerModeDesc(java.util.Locale locale) Create a `SynthesizerModeDesc` with a given `Locale` and other features set to `null`.
SynthesizerModeDesc(java.lang.String engineName, java.lang.String modeName, java.util.Locale locale, java.lang.Boolean running, Voice[] voices) Create a fully-specified descriptor.

Method Summary

void	**addVoice**(Voice v) Append a voice to the list of voices.
boolean	**equals**(java.lang.Object anObject) Returns true if and only if the parameter is not `null` and is a `SynthesizerModeDesc` with equal values of engine name, mode name, locale, running, and all voices.
Voice[]	**getVoices**() Returns the list of voices available in this synthesizer mode.
boolean	**match**(EngineModeDesc require) Determine whether a `SynthesizerModeDesc` has all the features specified by the `require` object.
void	**setVoices**(Voice[] v) Set the list of synthesizer voices.

Methods inherited from class javax.speech.EngineModeDesc
equals, getEngineName, getLocale, getModeName, getRunning, match, setEngineName, setLocale, setModeName, setRunning

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

SynthesizerModeDesc

`public SynthesizerModeDesc()`

Construct a descriptor with all features set to `null`.

SynthesizerModeDesc

`public SynthesizerModeDesc(java.util.Locale locale)`

Create a `SynthesizerModeDesc` with a given `Locale` and other features set to `null`.

SynthesizerModeDesc

```
public SynthesizerModeDesc(java.lang.String engineName,
                           java.lang.String modeName,
                           java.util.Locale locale,
                           java.lang.Boolean running,
                           Voice[] voices)
```

Create a fully-specified descriptor. Any of the features may be `null`.

Method Detail

getVoices

```
public Voice[] getVoices()
```

Returns the list of voices available in this synthesizer mode.

setVoices

```
public void setVoices(Voice[] v)
```

Set the list of synthesizer voices.

addVoice

```
public void addVoice(Voice v)
```

Append a voice to the list of voices.

match

```
public boolean match(EngineModeDesc require)
```

Determine whether a `SynthesizerModeDesc` has all the features specified by the `require` object. Features in `require` which are either `null` or zero-length strings ("") are not tested (including those contained by `Locale`). All string comparisons are exact (case-sensitive).

The parameters are used as follows:

- First, all features of the `EngineModeDesc` class are compared. If any test fails, the method returns false.
- If the require parameter is a `SynthesizerModeDesc` (or sub-class) then the required voice list is tested as follows.
  Each voice defined in the required set must match one of the voices in the tested object. (See `Voice.match()` for details.)

Note: if is possible to compare an `EngineModeDesc` against a `SynthesizerModeDesc` and vice versa.

Overrides:
match in class EngineModeDesc
See Also:
match, match

equals

```
public boolean equals(java.lang.Object anObject)
```

Returns true if and only if the parameter is not `null` and is a `SynthesizerModeDesc` with equal values of engine name, mode name, locale, running, and all voices.
Overrides:
equals in class EngineModeDesc

Interface javax.speech.synthesis.SynthesizerProperties

public abstract interface **SynthesizerProperties**
extends EngineProperties

Provides control of the run-time properties of a `Synthesizer`. The `SynthesizerProperties` object is obtained by calling the `getEngineProperties` method of the `Synthesizer` (inherited from the `Engine` interface).

Because `SynthesizerProperties` extends the `EngineProperties` interface to provide synthesizer-specific properties. It also inherits the following properties and conventions from the `EngineProperties` interface.

- Each property has a get and set method. (JavaBeans property method patterns)
- Engines may ignore calls to change properties, for example by applying maximum and minimum setttings.
- Calls to set methods may be asynchronous (they may return before the property change takes effect). The `Engine` will apply a change as soon as possible. A `PropertyChangeEvent` is issued when the change takes effect. For example, a change in the speaking rate might take place immediately, or at the end of the current word, sentence or paragraph.
- The get methods return the current setting - not a pending value.
- A `PropertyChangeListener` may be attached to receive property change events.
- Where appropriate, property settings are persistent across sessions.

The properties of a synthesizer are:

- Speaking voice,
- Baseline pitch,
- Pitch range,
- Speaking rate,
- Volume.

Setting these properties should be considered as a *hint* to the synthesizer. A synthesizer may choose to ignore out-of-range values. A synthesizer may have some properties that are unchangeable (e.g. a single voice synthesizer). Reasonable values for baseline pitch, pitch range and speaking rate may vary between synthesizers, between languages and or between voices.

A change in voice may lead to change in other properties. For example, female and young voices typically have higher pitches than male voices. When a change in voice leads to changes in other properties, a separate `PropertyChangeEvent` is issued for each property changed.

Whenever possible, property changes should be persistent for a voice. For instance, after changing from voice A to voice B and back, the previous property settings for voice A should return.

Changes in pitch, speaking rate and so on in the Java Speech Markup Language text provided to the synthesizer affect the get values but do not lead to a property change event. Applications needing an event at the time these changes should include a `MARK` property with the appropriate JSML element.

See Also:
Synthesizer, getEngineProperties

Method Summary

float	**getPitchRange**() Get the pitch range for synthesis.
float	**getPitch**() Get the baseline pitch for synthesis.
float	**getSpeakingRate**() Get the current target speaking rate.
Voice	**getVoice**() Get the current synthesizer voice.
float	**getVolume**() Get the current volume.
void	**setPitchRange**(float hertz) Set the pitch range for the current synthesis voice.
void	**setPitch**(float hertz) Set the baseline pitch for the current synthesis voice.
void	**setSpeakingRate**(float wpm) Set the target speaking rate for the synthesis voice in words per minute.
void	**setVoice**(Voice voice) Set the current synthesizer voice.
void	**setVolume**(float volume) Set the volume for the synthesizer's speech output as a value between 0.0 and 1.0. A value of 0.0 indicates silence.

Method Detail

getVoice

`public Voice getVoice()`

Get the current synthesizer voice. Modifications to the returned voice do not affect the `Synthesizer` voice - a call to `setVoice` is required for a change to take effect.

See Also:
setVoice

setVoice

```
public void setVoice(Voice voice)
             throws java.beans.PropertyVetoException
```

Set the current synthesizer voice.

The list of available voices for a `Synthesizer` is returned by the getVoices method of the synthesizer's `SynthesizerModeDesc`. Any one of the voices returned by that method can be passed to `setVoice` to set the current speaking voice.

Alternatively, the voice parameter may be an application-created partially specified `Voice` object. If there is no matching voice, the call is ignored. For example, to select a young female voice:

```
Voice voice = new Voice(null
                        GENDER_FEMALE,
                        AGE_CHILD | AGE_TEENAGER,
                        null);
synthesizerProperties.setVoice(voice);
```

Throws:
java.beans.PropertyVetoException - if the synthesizer rejects or limits the new value
See Also:
getVoice, SynthesizerModeDesc, getEngineModeDesc

---

getPitch

```
public float getPitch()
```

Get the baseline pitch for synthesis.

See Also:
setPitch

---

setPitch

```
public void setPitch(float hertz)
             throws java.beans.PropertyVetoException
```

Set the baseline pitch for the current synthesis voice. Out-of-range values may be ignored or restricted to engine-defined limits. 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.

Throws:
java.beans.PropertyVetoException - if the synthesizer rejects or limits the new value
See Also:
getPitch

---

getPitchRange

```
public float getPitchRange()
```

Get the pitch range for synthesis.

See Also:
setPitchRange

---

setPitchRange

```
public void setPitchRange(float hertz)
                  throws java.beans.PropertyVetoException
```

Set the pitch range for the current synthesis voice. A narrow pitch range provides monotonous output while wide range provide a more lively voice. This setting is a hint to the synthesis engine. Engines may choose to ignore unreasonable requests. Some synthesizers may not support pitch variability.

The pitch range is typically between 20% and 80% of the baseline pitch.

Throws:
java.beans.PropertyVetoException - if the synthesizer rejects or limits the new value
See Also:
getPitchRange

---

getSpeakingRate

```
public float getSpeakingRate()
```

Get the current target speaking rate.

See Also:
setSpeakingRate

---

setSpeakingRate

```
public void setSpeakingRate(float wpm)
                    throws java.beans.PropertyVetoException
```

Set the target speaking rate for the synthesis voice in words per minute.

Reasonable speaking rates depend upon the synthesizer and the current voice (some voices sound better at higher or lower speed than others).

Speaking rate is also dependent upon the language because of different conventions for what is a "word". A reasonable speaking rate for English is 200 words per minute.

Throws:
java.beans.PropertyVetoException - if the synthesizer rejects or limits the new value

See Also:
getSpeakingRate

---

getVolume

```
public float getVolume()
```

Get the current volume.

See Also:
setVolume

---

setVolume

```
public void setVolume(float volume)
               throws java.beans.PropertyVetoException
```

Set the volume for the synthesizer's speech output as a value between 0.0 and 1.0. A value of 0.0 indicates silence. A value of 1.0 is maximum volume and is usually the synthesizer default.

A synthesizer may change the voice's style with volume. For example, a quiet volume might produce whispered output and loud might produce shouting. Most synthesizer *do not* make this type of change.

Throws:
java.beans.PropertyVetoException - if the synthesizer rejects or limits the new value
See Also:
getVolume

Class javax.speech.synthesis.SynthesizerQueueItem

```
java.lang.Object
  |
  +--javax.speech.synthesis.SynthesizerQueueItem
```

public class **SynthesizerQueueItem**
extends java.lang.Object

Represents an object on the speech output queue of a `Synthesizer`. The item is described by the source object, the speakable text, a boolean value indicating whether it is a plain text object, and the `SpeakableListener` for the object.

The source object is the object provided to a `speak` method (a `Speakable` objects, a `URL`, or a `String`). The text is the Java Speech Markup Language string or plain text obtained from the source object. The listener is the `SpeakableListener` object passed to the `speak` method, or `null`.

The `enumerateQueue` method of a `Synthesizer` provides a snapshot of the speech output queue. It returns an enumeration object that is `null` if the queue is empty or contains a set of `SynthesizerQueueItems`. The `Synthesizer's` queue cannot be manipulated through this enumeration object.

See Also:
Synthesizer, enumerateQueue

Field Summary

SpeakableListener	**listener** The listener for this object passed to the `speak` method or `null`.
boolean	**plainText** True if the text object is plain text (not Java Speech Markup Language).
java.lang.Object	**source** The soure object for an item on the speech output queue.
java.lang.String	**text** The speakable text for an item on the speech output queue.

Constructor Summary

SynthesizerQueueItem(java.lang.Object source, java.lang.String text, boolean plainText, SpeakableListener listener) Construct a `SynthesizerQueueItem` with the source object and speakable text.

Method Summary

java.lang.Object	**getSource**() Return the source object for an item on the speech output queue of a `Synthesizer`.
SpeakableListener	**getSpeakableListener**() Return the `SpeakableListener` object for this speech output queue item, or `null` if none was provided to the `speak` method.
java.lang.String	**getText**() Return the speakable text for an item on the speech output queue of a `Synthesizer`.
boolean	**isPlainText**() Return `true` if the item contains plain text (not Java Speech Markup Language text).

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

source

`protected java.lang.Object` **`source`**

The soure object for an item on the speech output queue.

See Also:
getSource

---

text

`protected java.lang.String` **`text`**

The speakable text for an item on the speech output queue.

See Also:
getText

---

plainText

`protected boolean` **`plainText`**

True if the text object is plain text (not Java Speech Markup Language).

See Also:
isPlainText

---

listener

`protected SpeakableListener` **`listener`**

The listener for this object passed to the `speak` method or `null`.

See Also:
getSpeakableListener

Constructor Detail

SynthesizerQueueItem

```
public SynthesizerQueueItem(java.lang.Object source,
                            java.lang.String text,
                            boolean plainText,
                            SpeakableListener listener)
```

Construct a `SynthesizerQueueItem` with the source object and speakable text.

Method Detail

getSource

`public java.lang.Object` **`getSource`**`()`

Return the source object for an item on the speech output queue of a `Synthesizer`. The source is one of the three object types passed to the `speak` or `speakPlainText` methods of `Synthesizer`: a `Speakable` objects, a `URL`, or a `String`.

---

getText

`public java.lang.String` **`getText`**`()`

Return the speakable text for an item on the speech output queue of a `Synthesizer`. The text is either a Java Speech Markup Language string or a plain text string that was obtained from source object.

---

isPlainText

`public boolean` **`isPlainText`**`()`

Return `true` if the item contains plain text (not Java Speech Markup Language text).

---

getSpeakableListener

`public SpeakableListener` **`getSpeakableListener`**`()`

Return the `SpeakableListener` object for this speech output queue item, or `null` if none was provided to the `speak` method.

Class javax.speech.synthesis.Voice

```
java.lang.Object
  |
  +--javax.speech.synthesis.Voice
```

public class **Voice**
extends java.lang.Object
implements java.lang.Cloneable

A description of one output voice of a speech synthesizer. `Voice` objects can be used in selection of synthesis engines (through the `SynthesizerModeDesc`). The current speaking voice of a `Synthesizer` can be changed during operation with the setVoice method of the `SynthesizerProperties` object.

See Also:
SynthesizerModeDesc, setVoice, Synthesizer

Field Summary

static int	**AGE_CHILD** Age roughly up to 12 years.
static int	**AGE_DONT_CARE** Ignore age when performing a match.
static int	**AGE_MIDDLE_ADULT** Age roughly 40 to 60 years.
static int	**AGE_NEUTRAL** Voice with age that is indeterminate.
static int	**AGE_OLDER_ADULT** Age roughly 60 years and up.
static int	**AGE_TEENAGER** Age roughly 13 to 19 years.
static int	**AGE_YOUNGER_ADULT** Age roughly 20 to 40 years.
static int	**GENDER_DONT_CARE** Ignore gender when performing a match of voices.
static int	**GENDER_FEMALE** Female voice.
static int	**GENDER_MALE** Male voice.
static int	**GENDER_NEUTRAL** Neutral voice that is neither male or female (for example, artificial voices, robotic voices).

Constructor Summary

Voice() Empty constructor sets voice name and style to `null`, and age and gender to "don't care" values.
Voice(java.lang.String name, int gender, int age, java.lang.String style) Constructor provided with voice name, gender, age and style.

Method Summary

java.lang.Object	**clone**() Create a copy of this `Voice`.
boolean	**equals**(java.lang.Object anObject) Returns true if and only if the parameter is not `null` and is a `Voice` with equal values of name, age, gender, and style.
int	**getAge**() Get the voice age.
int	**getGender**() Get the voice gender.
java.lang.String	**getName**() Get the voice name.
java.lang.String	**getStyle**() Set the voice style.
boolean	**match**(Voice require) Determine whether a `Voice` has all the features defined in the `require` object.
void	**setAge**(int age) Set the voice age.
void	**setGender**(int gender) Set the voice gender.
void	**setName**(java.lang.String name) Set the voice name.
void	**setStyle**(java.lang.String style) Set the voice style.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

GENDER_DONT_CARE

`public static final int **GENDER_DONT_CARE**`

Ignore gender when performing a match of voices. Synthesizers never provide a voice with `GENDER_DONT_CARE`.

See Also:
getGender, setGender

GENDER_FEMALE

`public static final int **GENDER_FEMALE**`

Female voice.

See Also:
getGender, setGender

GENDER_MALE

`public static final int **GENDER_MALE**`

Male voice.

See Also:
getGender, setGender

GENDER_NEUTRAL

`public static final int **GENDER_NEUTRAL**`

Neutral voice that is neither male or female (for example, artificial voices, robotic voices).

See Also:
getGender, setGender

AGE_DONT_CARE

`public static final int **AGE_DONT_CARE**`

Ignore age when performing a match. Synthesizers never provide a voice with `AGE_DONT_CARE`.

See Also:
getAge, setAge

AGE_CHILD

`public static final int` **`AGE_CHILD`**

Age roughly up to 12 years.

See Also:
getAge, setAge

AGE_TEENAGER

`public static final int` **`AGE_TEENAGER`**

Age roughly 13 to 19 years.

See Also:
getAge, setAge

AGE_YOUNGER_ADULT

`public static final int` **`AGE_YOUNGER_ADULT`**

Age roughly 20 to 40 years.

See Also:
getAge, setAge

AGE_MIDDLE_ADULT

`public static final int` **`AGE_MIDDLE_ADULT`**

Age roughly 40 to 60 years.

See Also:
getAge, setAge

AGE_OLDER_ADULT

`public static final int` **`AGE_OLDER_ADULT`**

Age roughly 60 years and up.

See Also:
getAge, setAge

AGE_NEUTRAL

`public static final int` **`AGE_NEUTRAL`**

Voice with age that is indeterminate. For example, artificial voices, robotic voices.

See Also:
getAge, setAge

Constructor Detail

Voice

`public` **`Voice`**`()`

Empty constructor sets voice name and style to `null`, and age and gender to "don’t care" values.

Voice

```
public Voice(java.lang.String name,
             int gender,
             int age,
             java.lang.String style)
```

Constructor provided with voice name, gender, age and style.

Method Detail

getName

`public java.lang.String` **`getName`**`()`

Get the voice name. May return `null`.

---

setName

`public void` **`setName`**`(java.lang.String name)`

Set the voice name. A null or "" string in voice match means don’t care.

---

getGender

`public int` **`getGender`**`()`

Get the voice gender. Gender values are OR’able. For example, to test whether a voice is male and/or neutral:

```
Voice voice = ...;
Voice test = new Voice();
test.setGender(Voice.GENDER_MALE | Voice.GENDER_NEUTRAL);
if (voice.match(test)) ...
```

See Also:
GENDER_FEMALE, GENDER_MALE, GENDER_NEUTRAL, GENDER_DONT_CARE

setGender

`public void setGender(int gender)`

Set the voice gender.

See Also:
GENDER_FEMALE, GENDER_MALE, GENDER_NEUTRAL, GENDER_DONT_CARE

getAge

`public int getAge()`

Get the voice age. Age values are OR'able. For example, to test whether a voice is child or teenager (less than 20):

```
Voice voice = ...;
Voice test = new Voice();
test.setAge(Voice.AGE_CHILD | Voice.AGE_TEENAGER);
if (voice.match(test)) ...
```

See Also:
AGE_CHILD, AGE_TEENAGER, AGE_YOUNGER_ADULT, AGE_MIDDLE_ADULT, AGE_OLDER_ADULT, AGE_NEUTRAL, AGE_DONT_CARE

setAge

`public void setAge(int age)`

Set the voice age.

See Also:
AGE_CHILD, AGE_TEENAGER, AGE_YOUNGER_ADULT, AGE_MIDDLE_ADULT, AGE_OLDER_ADULT, AGE_NEUTRAL, AGE_DONT_CARE

getStyle

`public java.lang.String getStyle()`

Set the voice style. This parameter is designed for human interpretation. Values might include "business", "casual", "robotic", "breathy".

setStyle

`public void setStyle(java.lang.String style)`

Set the voice style.

clone

`public java.lang.Object clone()`

Create a copy of this `Voice`.
Returns:
A copy of this
Overrides:
clone in class java.lang.Object

match

`public boolean match(Voice require)`

Determine whether a `Voice` has all the features defined in the `require` object. Strings in `require` which are either `null` or zero-length ("") are ignored. All string comparisons are exact matches (case-sensitive).

`GENDER_DONT_CARE` and `AGE_DONT_CARE` values in the `require` object are ignored. The age and gender parameters are OR'ed: e.g. the required age can be `AGE_TEENAGER | AGE_CHILD`.

equals

`public boolean equals(java.lang.Object anObject)`

Returns true if and only if the parameter is not `null` and is a `Voice` with equal values of name, age, gender, and style.
Overrides:
equals in class java.lang.Object

Package javax.speech.recognition

Interface Summary

DictationGrammar	Provides access to the dictation capabilities of a `Recognizer`.
FinalDictationResult	Provides information on a finalized result for an utterance that matches a `DictationGrammar`.
FinalResult	`FinalResult` is an extension to the `Result` interface that provides information about a result that has been finalized - that is, recognition is complete.
FinalRuleResult	Provides information on a finalized result for an utterance that matches a `RuleGrammar`.
Grammar	Parent interface supported by all recognition grammars including `DictationGrammar` and `RuleGrammar`.
GrammarListener	A `GrammarListener` receives notifications of status change events for a `Grammar`.
RecognizerAudioListener	Extends the set of audio event of an engine for a recognizer by adding a audio level event.
Recognizer	A `Recognizer` provides access to speech recognition capabilities.
RecognizerListener	Defines an extension to the `EngineListener` interface for specific events associated with a `Recognizer`.
RecognizerProperties	Enables control of the properties of a `Recognizer`.
Result	A `Result` is issued by a `Recognizer` as it recognizes an incoming utterance that matches an active `Grammar`.
ResultListener	The methods of a `ResultListener` receive notifications of events related to a `Result` object.
ResultToken	A token (usually a word) contained by a `Result` representing something heard by a recognizer.
RuleGrammar	`RuleGrammar` interface describes a `Grammar` that defines what users may say by a set rules.
SpeakerManager	Provides control of `SpeakerProfiles` for a `Recognizer`.

Class Summary

GrammarAdapter	The adapter which receives grammar events.
GrammarEvent	A `GrammarEvent` is issued to each `GrammarListener` attached to a `Grammar` when major events associated with that `Grammar` occur.
GrammarSyntaxDetail	Description of a problem found in a grammar usually bundled with a `GrammarException`.
RecognizerAdapter	The adapter which receives events for a `Recognizer`.
RecognizerAudioAdapter	Adaptor for a audio events of a recognizer.
RecognizerAudioEvent	Event issued to indicate detection of speech in the incoming audio stream or to periodically indicate the audio input level.
RecognizerEvent	Event issued by `Recognizer` through `RecognizerListener`.
RecognizerModeDesc	`RecognizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech recognizers.
ResultAdapter	The adapter which receives events for a `Result` object.
ResultEvent	A `ResultEvent` is issued by a `Result` object to indicate changes in the recognized tokens and changes in state.
RuleAlternatives	`RuleAlternatives` represents a `Rule` composed of a set of alternative sub-rules.
RuleCount	Attaches a count to a contained `Rule` object to indicate the number of times it may occur.
Rule	A `Rule` object is the basic component of a `RuleGrammar` and represents anything that may appear on the right-hand side of a rule definition in Java Speech Grammar Format.
RuleName	A `RuleName` is a reference to a named rule.
RuleParse	Represents the output of a parse of a `Result` or a string against a `RuleGrammar`.
RuleSequence	`RuleSequence` is a `Rule` composed of a sequence of sub-rules that must each be spoken in order.
RuleTag	`RuleTag` attaches a tag to a contained `Rule` object.
RuleToken	`RuleToken` represents speakable text in a `RuleGrammar`.
SpeakerProfile	A `SpeakerProfile` object is used to identify each enrollment by a user to a `Recognizer`.

Exception Summary

GrammarException	Thrown if a problem is found with a Java Speech Grammar Format (JSGF) file or with a `RuleGrammar` object derived from JSGF. Grammar problems are typically identified and fixed during application development.

Error Summary

ResultStateError	Signals an error caused by an illegal call to a method of `FinalResult`, `FinalRuleResult` or `FinalDictationResult`.

Interface javax.speech.recognition.DictationGrammar

public abstract interface **DictationGrammar**
extends Grammar

Provides access to the dictation capabilities of a `Recognizer`. If a recognizer supports dictation, it provides a `DictationGrammar` which is obtained through the `getDictationGrammar` method of the `Recognizer`.

A `DictationGrammar` is named with the same convention as a `RuleGrammar` and will typically reflect the language and domain it supports. The grammar name is obtained through the `Grammar.getName` method. For example, the general general dictation for US English from Acme speech company might be called:

```
com.acme.dictation.english.us.general
```

A `DictationGrammar` is characterized by:

- Typically large vocabulary.
- Grammar is built-into the recognizer,
- General purpose or domain-specific (e.g. legal, radiology, medical),
- May support continuous or discrete speech,

A dictation grammar is built into a recognizer (if supported). Moreover, a recognizer provides a single dictation grammar. Applications cannot create or directly modify the grammar beyond the relatively simple methods provided by this interface. By comparison, an application can change any part of a `RuleGrammar`. (Some vendors provide tools for constructing dictation grammars, but these tools operate separate from the Java Speech API.)

A recognizer that supports dictation:

- Returns `true` for the `RecognizerModeDesc.isDictationGrammarSupported` method This value can be used to select a dictation recognizer.
- Typically resource intensive (CPU, disk, memory),
- Often requires training by user (supports the `SpeakerManager` interface).

DictationGrammar Extends Grammar

The `DictationGrammar` interface extends the `Grammar` interface. Thus, a `DictationGrammar` provides all the `Grammar` functionality:

- The dictation grammar name is returned by the `Grammar.getName` method.
- The dictation grammar is enabled and disabled for recognition by the `setEnabled` method inherited from `Grammar`.
- The activation mode is set through the `setActivationMode` method of `Grammar`. Note: a `DictationGrammar` should *never* have `GLOBAL` activation mode.
- The current activation state of the `Grammar` is tested by the `isActive` method and the `GRAMMAR_ACTIVATED` and `GRAMMAR_DEACTIVATED` events.
- Grammar listeners are attached and removed by the `Grammar.addGrammarListener` and `Grammar.removeGrammarListener` methods.

Context

Dictation recognizers can use the current textual context to improve recognition accuracy. Applications should use either of the `setContext` methods to inform the recognizer each time the context changes. For example, when editing *this* sentence with the cursor after the word "sentence", the preceding context is "When editing this sentence" and the following context is "with the cursor after...". Any time the text context changes, the application should inform the recognizer.

Applications should provide a minimum of 3 or 4 words of context (when it's available). Different recognizers process context differently. Some recognizers can take advantage of several paragraphs of context others look only at a few words.

One form of `setContext` takes one string each for preceding and following context (as in the example above). The other form of `setContext` takes an array each of strings for preceding and following context. Arrays should be provided if the surrounding context is made of tokens from previous recognition results. When possible, providing tokens is the preferred method because recognizers are able to use token information more reliably and more efficiently.

Word Lists

Words can be added to and removed from the `DictationGrammar` using the `addWord` and `removeWord` methods. Lists of the added and removed words are available through the `listAddedWords` and `listRemovedWords` methods. In a speaker-adaptive system (`SpeakerManager` supported) word data is stored as part of the speaker's data.

Adding a word allows a recognizer to learn new words. Removing a word is useful when a recognizer consistently misrecognizes similar sounding words. For example, if each time a user says "grammar", the recognizer hears "grandma", the user can remove grandma (assuming they don't want to use the word "grandma").

See Also:
SpeakerManager, Recognizer, Grammar

Method Summary

void	**addWord**(java.lang.String word) Add a word to the `DictationGrammar`.
java.lang.String[]	**listAddedWords**() List the words that have been added to the `DictationGrammar`.
java.lang.String[]	**listRemovedWords**() List the words that have been removed from the `DictationGrammar`.
void	**removeWord**(java.lang.String word) Remove a word from the `DictationGrammar`.
void	**setContext**(java.lang.String preceding, java.lang.String following) Provide the recognition engine with the current textual context.
void	**setContext**(java.lang.String[] preceding, java.lang.String[] following) Provide the recognition engine with the current textual context with arrays of the previous and following tokens.

Method Detail

setContext

```
public void setContext(java.lang.String preceding,
                       java.lang.String following)
```

Provide the recognition engine with the current textual context. Dictation recognizers use the context information to improve recognition accuracy. (Context is discussed above.)

When dictating a sequence of words, the recognizer updates its context. The app does not need to inform the recognizer when results arrive. Instead it should call `setContext` for events such as cursor movement, cut, paste etc.

The preceding or following context may be `null` if there is no preceding or following context. This is appropriate for a new document or in situations where context is not clear.

The alternative `setContext`, that accepts arrays of tokens for context, should be used when the current context includes tokens from previous results.

See Also:
setContext(String[], String[])

---

setContext

```
public void setContext(java.lang.String[] preceding,
                       java.lang.String[] following)
```

Provide the recognition engine with the current textual context with arrays of the previous and following tokens. Dictation recognizers use the context information to improve recognition accuracy. (Context is discussed above.)

When dictating a sequence of words, the recognizer updates its context. The app does not need to inform the recognizer when results arrive. Instead it should call `setContext` for events such as cursor movement, cut, paste etc.

The preceding or following context may be `null` if there is no preceding or following context. This is appropriate for a new document or in situations where context is not clear.

See Also:
setContext(String, String)

---

addWord

```
public void addWord(java.lang.String word)
```

Add a word to the `DictationGrammar`. The `addWord` method can undo the effects of an `removeWord` call.

A change in a `DictationGrammar` is applied to the recognition process only after changes have been committed.

See Also:
commitChanges

---

removeWord

```
public void removeWord(java.lang.String word)
```

Remove a word from the `DictationGrammar`. The `removeWord` method can undo the effects of an `addWord` call.

A change in a `DictationGrammar` is applied to the recognition process only after changes have been committed.

See Also:
commitChanges

---

listAddedWords

```
public java.lang.String[] listAddedWords()
```

List the words that have been added to the `DictationGrammar`.

---

listRemovedWords

```
public java.lang.String[] listRemovedWords()
```

List the words that have been removed from the `DictationGrammar`.

Interface javax.speech.recognition.FinalDictationResult

public abstract interface **FinalDictationResult**
extends FinalResult

Provides information on a finalized result for an utterance that matches a `DictationGrammar`. A finalized result is a result that is in either the `ACCEPTED` or `REJECTED` state (tested by the `getResultState` method of the `Result` interface).

The `FinalDictationResult` interface extends the `Result` and `FinalResult` interfaces with a single method. The `getAlternativeTokens` method provides access to alternative guesses for tokens in a dictation result.

Every `Result` object provided by a `Recognizer` implements the `FinalDictationResult` and the `FinalRuleResult` interfaces (and by inheritence the `FinalResult` and `Result` interfaces). However, the methods of `FinalDictationResult` should only be called if the `Result.getGrammar` method returns a `DictationGrammar` and once the `Result` has been finalized with either an `RESULT_ACCEPTED` or `RESULT_REJECTED` event. Inappropriate calls will cause a `ResultStateError`.

See Also:
ACCEPTED, REJECTED, getResultState, DictationGrammar, Result, FinalResult, FinalRuleResult, ResultStateError

---

Method Summary

ResultToken[][]	**getAlternativeTokens**(ResultToken fromToken, ResultToken toToken, int max) Return a set of alternative token guesses for a single known token or sequence of tokens.

Method Detail

getAlternativeTokens

```
public ResultToken[][] getAlternativeTokens(ResultToken fromToken,
                                            ResultToken toToken,
                                            int max)
                                     throws ResultStateError,
                                            java.lang.IllegalArgumentException
```

Return a set of alternative token guesses for a single known token or sequence of tokens. In a dictation application the alternative guesses are typically provided to a user to facilitate efficient correction of dictated text. The assumption is that if a recognizer does not correctly hear a user (a mis-recognition) the correct tokens are likely to be amongst the top alternatives guesses.

Typically when a user selects one of the alternative guesses as the correct token sequence, the `tokenCorrection` method of the `FinalResult` interface is called with those tokens. That call allows the `Recognizer` to learn from recognition errors and can improve future recognition accuarcy.

The `fromToken` and `toToken` parameters define an inclusive set of tokens for which alternatives are required. `fromToken` and `toToken` are typically included in the set of alternative guesses. If `toToken` is `null` or if `fromToken` and `toToken` are the same, then alternatives are provided for the single `toToken` token.

The tokens passed to this method must be `ResultToken` objects provided by this result through previous calls to `getAlternativeTokens` or the `getBestToken` and `getBestTokens` methods of the `Result` interface.

Returned Array Structure

The return value is a *ragged two-dimension array* with indices being `altTokens[guessNumber][tokenNumber]`. The guesses are ordered by from best guess to least likely (determined by the recognizer): `altTokens[0]` is the best guess for the span of `fromToken` to `toToken`, `altTokens[1]` is the first alternative guess and so on.

The number of tokens may be different in each alternative guess. This means that the length of `altTokens[0]` may be different from the length of `altTokens[1]` and so on. The length is never zero or less. (This point is illustrated in the example below.)

The `max` parameter indicates the maximum number of alternative guesses to be returned. The number of guesses returned may be less than or equal to `max`. The number of alternatives returned is also less than or equal to the `NumResultAlternatives` property set in the `RecognizerProperties` at the time of recognition.

The number of alternative guesses and the number of tokens in each guess can vary between results. The numbers can vary for different values of `fromToken` and `toToken` for the calls to the same result.

The returned alternative guess is always an array of length one or greater. If there is only one guess, it may be the sequence of `fromToken` to `toToken` tokens. Each guess always contains one or more tokens. If the result is `ACCEPTED` then the recognizer is confident that all the alternatives guesses are reasonable guesses of what the user said.

Example

Assume the user says "You can recognize speech" and that the recognizer accepts a result and hears the utterance correctly. The result will have four tokens (`Result.numTokens() == 4`) and the best guess tokens returned by the `Result.getBestTokens()` method returns and array with `{"you", "can", "recognize", "speech"}`. (The `Result.getBestToken(int index)` method will return the tokens one at a time.)

Let’s take the second and last tokens ("can" and "speech") and request five alternative guesses for the sequence including those tokens:

```
FinalDictationResult result;
ResultToken secondToken = result.getBestToken(1);
ResultToken lastToken = result.getBestToken(result.numTokens());
ResultToken[][] altTokens = result.getAlternativeTokens(secondToken, lastToken, 5);
```

The return value might look like the following (representing each token by its written-form

string):

```
altTokens = { {"can", "recognize", "speech"},
              {"can", "wreck", "a", "nice", "beach"},
              {"in", "recognize", "beach"},
              {"recognize", "speech"} }
```

Some observations from this example which illustrate the points described above.

- Only 4 guesses are returned, less than 5 guesses requested in the `getAlternativeTokens` call.
- The first token sequence is the same as the best guess as returned by the `getBestTokens` method. (As we expected, "can recognize speech" is the recognizer’s best guess for that segment of incoming speech.)
- The number of tokens is different between guesses: 3, 5, 3, 2 tokens in each.

Throws:
ResultStateError - if called before a result is finalized or if the matched grammar is not a `DictationGrammar`
java.lang.IllegalArgumentException - if passed a `ResultToken` not obtained from this result
See Also:
Result, numTokens, getBestToken, getBestTokens, tokenCorrection, setNumResultAlternatives

---

Interface javax.speech.recognition.FinalResult

Subinterfaces:
FinalDictationResult, FinalRuleResult

---

public abstract interface **FinalResult**
extends Result

`FinalResult` is an extension to the `Result` interface that provides information about a result that has been finalized - that is, recognition is complete. A finalized result is a `Result` that has received either a `RESULT_ACCEPTED` or `RESULT_REJECTED` `ResultEvent` that puts it in either the `ACCEPTED` or `REJECTED` state (indicated by the `getResultState` method of the `Result` interface).

The `FinalResult` interface provides information for finalized results that match either a `DictationGrammar` or a `RuleGrammar`.

Any result object provided by a recognizer implements both the `FinalRuleResult` and `FinalDictationResult` interfaces. Because both these interfaces extend the `FinalResult` interface, which in turn extends the `Result` interface, all results implement `FinalResult`.

The methods of the `FinalResult` interface provide information about a *finalized* result (`ACCEPTED` or `REJECTED` state). If any method of the `FinalResult` interface is called on a result in the `UNFINALIZED` state, a `ResultStateError` is thrown.

Three capabilities can be provided by a finalized result: training/correction, access to audio data, and access to alternative guesses. All three capabilities are optional because they are not all relevant to all results or all recognition environments, and they are not universally supported by speech recognizers. Training and access to audio data are provided by the `FinalResult` interface. Access to alternative guesses is provided by the `FinalDictationResult` and `FinalRuleResult` interfaces (depending upon the type of grammar matched).

Training / Correction

Because speech recognizers are not always correct, applications need to consider the possibility that a recognition error has occurred. When an application detects an error (e.g. a user updates a result), the application should inform the recognizer so that it can learn from the mistake and try to improve future performance. The `tokenCorrection` is provided for an application to provide feedback from user correction to the recognizer.

Sometimes, but certainly not always, the correct result is selected by a user from amongst the N-best alternatives for a result obtained through either the `FinalRuleResult` or `FinalDictationResult` interfaces. In other cases, a user may type the correct result or the application may infer a correction from following user input.

Recognizers must store considerable information to support training from results. Applications need to be involved in the management of that information so that it is not stored unnecessarily. The `isTrainingInfoAvailable` method tests whether training information is available for a finalized result. When an application/user has finished correction/training for a result it should call `releaseTrainingInfo` to free up system resources. Also, a recognizer may choose at any time to free up training information. In both cases, the application is notified of the the release with a `TRAINING_INFO_RELEASED` event to `ResultListeners`.

Audio Data

Audio data for a finalized result is optionally provided by recognizers. In dictation systems, audio feedback to users can remind them of what they said and is useful in correcting and proof-reading documents. Audio data can be stored for future use by an application or user and in certain circumstances can be provided by one recognizer to another.

Since storing audio requires substantial system resources, audio data requires special treatment. If an application wants to use audio data, it should set the `setResultAudioProvided` property of the `RecognizerProperties` to `true`.

Not all recognizers provide access to audio data. For those recognizers, `setResultAudioProvided` has no effect, the `FinalResult.isAudioAvailable` always returns `false`, and the `getAudio` methods always return `null`.

Recognizers that provide access to audio data cannot always provide audio for every result. Applications should test audio availability for every `FinalResult` and should always test for `null` on the `getAudio` methods.

See Also:
Result, getResultState, ACCEPTED, REJECTED, FinalDictationResult, FinalRuleResult, DictationGrammar, RuleGrammar, setResultAudioProvided

---

Field Summary

static int	**DONT_KNOW** The `DONT_KNOW` flag is used in a call to `tokenCorrection` to indicate that the application does not know whether a change to a result is because of `MISRECOGNITION` or `USER_CHANGE`.
static int	**MISRECOGNITION** The `MISRECOGNITION` flag is used in a call to `tokenCorrection` to indicate that the change is a correction of an error made by the recognizer.
static int	**USER_CHANGE** The `USER_CHANGE` flag is used in a call to `tokenCorrection` to indicate that the user has modified the text that was returned by the recognizer to something different from what they actually said.

Method Summary

java.applet.AudioClip	**getAudio**() Get the result audio for the complete utterance of a `FinalResult`.
java.applet.AudioClip	**getAudio**(ResultToken fromToken, ResultToken toToken) Get the audio for a token or sequence of tokens.
boolean	**isAudioAvailable**() Test whether result audio data is available for this result.
boolean	**isTrainingInfoAvailable**() Returns `true` if the `Recognizer` has training information available for this result.
void	**releaseAudio**() Release the result audio for the result.
void	**releaseTrainingInfo**() Release training information for this `FinalResult`.
void	**tokenCorrection**(java.lang.String[] correctTokens, ResultToken fromToken, ResultToken toToken, int correctionType) Inform the recognizer of a correction to one of more tokens in a finalized result so that the recognizer can re-train itself.

Field Detail

MISRECOGNITION

```
public static final int MISRECOGNITION
```

The `MISRECOGNITION` flag is used in a call to `tokenCorrection` to indicate that the change is a correction of an error made by the recognizer.

See Also:
tokenCorrection

USER_CHANGE

```
public static final int USER_CHANGE
```

The `USER_CHANGE` flag is used in a call to `tokenCorrection` to indicate that the user has modified the text that was returned by the recognizer to something different from what they actually said.

See Also:
tokenCorrection

DONT_KNOW

```
public static final int DONT_KNOW
```

The `DONT_KNOW` flag is used in a call to `tokenCorrection` to indicate that the application does not know whether a change to a result is because of `MISRECOGNITION` or `USER_CHANGE`.

See Also:
tokenCorrection

Method Detail

isTrainingInfoAvailable

```
public boolean isTrainingInfoAvailable()
                                throws ResultStateError
```

Returns `true` if the `Recognizer` has training information available for this result. Training is available if the following conditions are met:

- The `isTrainingProvided` property of the `RecognizerProperties` is set to `true`.
- And, the training information for this result has not been released by the application or by the recognizer. (The `TRAINING_INFO_RELEASED` event has not been issued.)

Calls to `tokenCorrection` have no effect if the training information is not available.

Throws:
ResultStateError - if called before a result is finalized
See Also:
setTrainingProvided(boolean), releaseTrainingInfo()

---

releaseTrainingInfo

```
public void releaseTrainingInfo()
                         throws ResultStateError
```

Release training information for this `FinalResult`. The release frees memory used for the training information -- this information can be substantial.

It is not an error to call the method when training information is not available or has already been released.

This method is asynchronous - the training info is not necessarily released when the call returns. A `TRAINING_INFO_RELEASED` event is issued to the `ResultListener` once the information is released. The `TRAINING_INFO_RELEASED` event is also issued if the recognizer releases the training information for any other reason (e.g. to reclaim memory).

Throws:
ResultStateError - if called before a result is finalized
See Also:
TRAINING_INFO_RELEASED

tokenCorrection

```
public void tokenCorrection(java.lang.String[] correctTokens,
                            ResultToken fromToken,
                            ResultToken toToken,
                            int correctionType)
                     throws ResultStateError,
                            java.lang.IllegalArgumentException
```

Inform the recognizer of a correction to one of more tokens in a finalized result so that the recognizer can re-train itself. Training the recognizer from its mistakes allows it to improve its performance and accuracy in future recognition.

The `fromToken` and `toToken` parameters indicate the inclusive sequence of best-guess or alternative tokens that are being trained or corrected. If `toToken` is `null` or if `fromToken` and `toToken` are the same, the training applies to a single recognized token.

The `correctTokens` token sequence may have the same of a different length than the token sequence being corrected. Setting `correctTokens` to `null` indicates the deletion of tokens.

The `correctionType` parameter must be one of `MISRECOGNITION`, `USER_CHANGE`, `DONT_KNOW`.

Note: `tokenCorrection` does not change the result object. So, future calls to the `getBestToken`, `getBestTokens` and `getAlternativeTokens` method return exactly the same values as before the call to `tokenCorrection`.

Parameters:
`correctTokens` - sequence of correct tokens to replace `fromToken` to `toToken`
`fromToken` - first token in the sequence being corrected
`toToken` - last token in the sequence being corrected
`correctionType` - type of correction: `MISRECOGNITION`, `USER_CHANGE`, `DONT_KNOW`
Throws:
java.lang.IllegalArgumentException - either token is not from this `FinalResult`
ResultStateError - if called before a result is finalized
See Also:
MISRECOGNITION, USER_CHANGE, DONT_KNOW

isAudioAvailable

```
public boolean isAudioAvailable()
                         throws ResultStateError
```

Test whether result audio data is available for this result. Result audio is only available if:

- The `ResultAudioProvided` property of `RecognizerProperties` was set to `true` when the result was recognized.
- The `Recognizer` was able to collect result audio for the current type of `FinalResult` (`FinalRuleResult` or `FinalDictationResult`).
- The result audio has not yet been released.

The availability of audio for a result does not mean that all `getAudio` calls will return an `AudioClip`. For example, some recognizers might provide audio data only for the entire result or only for individual tokens, or not for sequences of more than one token.

Throws:
ResultStateError - if called before a result is finalized
See Also:
getAudio, setResultAudioProvided

releaseAudio

```
public void releaseAudio()
                  throws ResultStateError
```

Release the result audio for the result. After audio is released, `isAudioAvailable` will return `false`. This call is ignored if result audio is not available or has already been released.

This method is asynchronous - audio data is not necessarily released immediately. A `AUDIO_RELEASED` event is issued to the `ResultListener` when the audio is released by a call to this method. A `AUDIO_RELEASED` event is also issued if the recognizer releases the audio for some other reason (e.g. to reclaim memory).

Throws:
ResultStateError - if called before a result is finalized
See Also:
AUDIO_RELEASED, audioReleased

getAudio

```
public java.applet.AudioClip getAudio()
                               throws ResultStateError
```

Get the result audio for the complete utterance of a `FinalResult`. Returns `null` if result audio is not available or if it has been released.

Throws:
ResultStateError - if called before a result is finalized
See Also:
isAudioAvailable, getAudio(ResultToken, ResultToken)

getAudio

```
public java.applet.AudioClip getAudio(ResultToken fromToken,
                          ResultToken toToken)
                              throws java.lang.IllegalArgumentException,
                                     ResultStateError
```

Get the audio for a token or sequence of tokens. Recognizers make a best effort at determining the start and end of tokens, however, it is not unusual for chunks of surrounding audio to be included or for the start or end token to be chopped.

Returns `null` if result audio is not available or if it cannot be obtained for the specified sequence of tokens.

If `toToken` is `null` or if `fromToken` and `toToken` are the same, the method returns audio for `fromToken`. If both `fromToken` and `toToken` are `null`, it returns the audio for the entire result (same as `getAudio()`).

Not all recognizers can provide per-token audio, even if they can provide audio for a complete utterance.

Throws:
java.lang.IllegalArgumentException - one of the token parameters is not from this result
ResultStateError - if called before a result is finalized

See Also:
getAudio()

Interface javax.speech.recognition.FinalRuleResult

public abstract interface **FinalRuleResult**
extends FinalResult

Provides information on a finalized result for an utterance that matches a `RuleGrammar`. A finalized result is one that is in either the `ACCEPTED` or `REJECTED` state (tested by the `getResultState` method of the `Result` interface).

This interface provides the following information for a finalized result for a `RuleGrammar` (in addition to the information provided through the `Result` and `FinalResult` interfaces):

- N-best alternatives for the entire result. For each alternative guess the interface provides the token sequence and the names of the grammar and rule matched by the tokens.
- Tags for the best-guess result.

Every `Result` object provided by a `Recognizer` implements the `FinalRuleResult` and the `FinalDictationResult` interfaces (and by inheritence the `FinalResult` and `Result` interfaces). However, the methods of `FinalRuleResult` should only be called if the `Result.getGrammar` method returns a `RuleGrammar` and once the `Result` has been finalized with either an `RESULT_ACCEPTED` or `RESULT_REJECTED` event. Inappropriate calls will cause a `ResultStateError`.

See Also:
RuleGrammar, Result, FinalResult, FinalDictationResult, getResultState, getGrammar

Method Summary

ResultToken[]	**getAlternativeTokens**(int nBest) Get the N-best token sequence for this result.
int	**getNumberGuesses**() Return the number of guesses for this result.
RuleGrammar	**getRuleGrammar**(int nBest) Return the `RuleGrammar` matched by the nth guess.
java.lang.String	**getRuleName**(int nBest) Return the `RuleName` matched by the nth guess.
java.lang.String[]	**getTags**() Return the list of tags matched by the best-guess token sequence.

Method Detail

getNumberGuesses

```
public int getNumberGuesses()
                  throws ResultStateError
```

Return the number of guesses for this result. The guesses are numbered from 0 up. The 0th guess is the best guess for the result and provides the same tokens as the `getBestTokens` method of the `Result` interface.

If only the best guess is available (no alternatives) the return value is 1. If the result is was rejected (`REJECTED` state), the return value may be 0 if no tokens are available. If a best guess plus alternatives are available, the return value is greater than 1.

The integer parameter to the `getAlternativeTokens`, `getRuleGrammar` and `getRuleName` methods are indexed from `0` to `(getNumberGuesses()-1)`.

Throws:
ResultStateError - if called before a result is finalized
See Also:
getBestTokens, REJECTED, getAlternativeTokens, getRuleGrammar, getRuleName

getAlternativeTokens

```
public ResultToken[] getAlternativeTokens(int nBest)
                                                    throws ResultStateError
```

Get the N-best token sequence for this result. The `nBest` value should be in the range of `0` to `(getNumberGuesses()-1)` inclusive. For out-of-range values, the method returns `null`.

If `nBest==0`, the method returns the best-guess sequence of tokens - identical to the token sequence returned by the `getBestTokens` method of the `Result` interface for the same object.

If `nBest==1` (or 2, 3...) the method returns the 1st (2nd, 3rd...) alternative to the best guess.

The number of tokens for the best guess and the alternatives do not need to be the same.

The `getRuleGrammar(int)` and `getRuleName(int)` methods indicate which `RuleGrammar` and `ruleName` are matched by each alternative result sequence.

If the `Result` is in the `ACCEPTED` state (not rejected), then the best guess and all the alternatives are accepted. Moreover, each alternative set of tokens must be a legal match of the `RuleGrammar` and `RuleName` returned by the `getRuleGrammar` and `getRuleName` methods.

If the `Result` is in the `REJECTED` state (not accepted), the recognizer is not confident that the best guess or any of the alternatives are what the user said. Rejected guesses do not need to match the corresponding `RuleGrammar` and rule name.

Example

Say we have two simple grammars loaded and active for recognition. The first is `grammar.numbers` with a public rule `digits` that matches spoken digit sequences (e.g. "1 2 3 4"). The grammar is `grammar.food` with the rule `whoAteWhat` which matches statements about eating (e.g. "he ate mine"). [Yes, this is artificial!]

The user says "two eight nine" and the recognizer correctly recognizes the speech, but also provides 2 alternatives. The

```
FinalRuleResult result = ...;
result.getNumberGuesses() -> 3

result.getAlternativeTokens(0) -> two eight nine // array of tokens
result.getRuleGrammar(0) -> [reference to grammar.numbers]
result.getRuleName(0) -> "digits"

result.getAlternativeTokens(1) -> you ate mine
result.getRuleGrammar(1) -> [reference to grammar.food]
result.getRuleName(1) -> "whoAteWhat"

result.getAlternativeTokens(2) -> two eight five
result.getRuleGrammar(2) -> [reference to grammar.numbers]
result.getRuleName(2) -> "digits"
```

Throws:
ResultStateError - if called before a result is finalized
See Also:
getRuleGrammar, getRuleName

getRuleGrammar

```
public RuleGrammar getRuleGrammar(int nBest)
                                            throws ResultStateError
```

Return the `RuleGrammar` matched by the nth guess. Return `null` if `nBest` is out-of-range.

`getRuleName` returns the rule matched in the `RuleGrammar`. See the documentation for `getAlternativeTokens` for a description of how tokens, grammars and rules correspond.

An application can use the parse method of the matched grammar to analyse a result. e.g.

```
int nBest = 2;
FinalRuleResult res;
RuleParse parse = res.getRuleGrammar(nbest).parse(
                          res.getAlternativeTokens(nBest),
                          res.getRuleName(nBest));
```

Throws:
ResultStateError - if called before a result is finalized
See Also:
parse, getNumberGuesses, getAlternativeTokens, getRuleName

getRuleName

```
public java.lang.String getRuleName(int nBest)
                               throws ResultStateError
```

Return the `RuleName` matched by the nth guess. Return `null` if `nBest` is out-of-range. Typically used in combination with `getAlternativeTokens` and `getRuleGrammar` which return the corresponding tokens and grammar.

The documentation for `getAlternativeTokens` shows and example result with alternatives.

Throws:
ResultStateError - if called before a result is finalized
See Also:
parse, getAlternativeTokens, getRuleGrammar

---

getTags

```
public java.lang.String[] getTags()
                             throws ResultStateError
```

Return the list of tags matched by the best-guess token sequence. The tags in the array are ordered strictly in the left to right order in which they would appear in JSGF.

For example, if the following simple Java Speech Grammar Format (JSGF) rule is active:

```
public  = (open {ACT_OPEN} | close {ACT_CLOSE}) [(it{WHAT} now) {NOW}];
```

and the user says "close it now", then getTags returns an array containing `{"ACT_CLOSE", "WHAT", "NOW"}`. Note how both the `{WHAT}` and `{NOW}` tags are attached starting from the word "it" but that `{TAG}` appears first in the array. In effect, when tags start at the same token, they are listed "bottom-up".

`getTags` does not indicate which tokens are matched by which tags. To obtain this information use the `parse` method of the `RuleGrammar`. Also, `getTags` only provides tags for the best guess. To get tags for the alternative guesses using parsing through the `RuleGrammar`.

The string array returned by the `getTags` method of the `RuleParse` object returned by parsing the best-guess token sequence will be the same as returned by this method.

Throws:
ResultStateError - if called before a result is finalized
See Also:
parse, getTags

Interface javax.speech.recognition.Grammar

Subinterfaces:
DictationGrammar, RuleGrammar

public abstract interface **Grammar**

Parent interface supported by all recognition grammars including `DictationGrammar` and `RuleGrammar`. A `Grammar` defines a set of tokens (words) that may be spoken and the patterns in which those tokens may be spoken. Different grammar types (dictation vs. rule) define the words and the patterns in different ways.

The core functionality provided through the `Grammar` interface includes:

- Naming: each grammar of a `Recognizer` has a unique name.
- Enabling: grammars may be enabled or disabled for activation of recognition.
- Activation: the activation mode can be set and the activation state tested.
- Adding and removing `GrammarListeners` for grammar-related events.
- Adding and removing `ResultListeners` to receive result events for results matching the `Grammar`.
- Access the `Recognizer` that owns the grammar

Each `Grammar` is associated with a specific `Recognizer`. All grammars are located, created and managed through the `Recognizer` interface. The basic steps in using any `Grammar` are:

1. Create a new `Grammar` or get a reference to an existing grammar through the `Recognizer` interface.
2. Attach a `ResultListener` to get results.
3. As necessary, setup or modify the grammar for the application’s context.
4. Enabled and disable recognition of the `Grammar` as required by the application’s context.
5. Commit grammar changes and enabled state into the recognition process.
6. Repeat the update, enable and commit steps as required.
7. For application-created grammars, delete the `Grammar` when it is no longer needed.

Grammar Types

There are two types of `Grammar`: `DictationGrammar` and `RuleGrammar` Both are defined by interfaces that extend the `Grammar` interface. All recognizers support `RuleGrammars`. A recognizer may optionally support a `DictationGrammar`.

The `RuleGrammar` and `DictationGrammar` interfaces define specialized mechanisms for controlling and modifying those types of grammar.

Grammar Activation

When a `Grammar` is active, the `Recognizer` listens to incoming audio for speech that matches that `Grammar`. To be activated, an application must first set the `enabled` property to `true` (enable activation) and set the `activationMode` property to indicate the *activation conditions*.

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 conditions are determined by `Recognizer` focus and possibly by the activation of other grammars. Recognizer focus is managed with the requestFocus and releaseFocus methods of a `Recognizer`.

The modes are listed here by priority. Always use the lowest priority mode possible.

- `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 `Recognizer` of this application has focus. Furthermore, enabling a modal grammar deactivates any grammars in the same `Recognizer` with the `RECOGNIZER_FOCUS` activation mode.
- `RECOGNIZER_FOCUS` activation mode (default): 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`.

An application may have zero, one or many grammar enabled 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`. However, when a `Recognizer` is `PAUSED`, audio input to the `Recognizer` is turned off, so speech can't be detected. Note that just after pausing a recognizer there may be some remaining audio in the input buffer that could contain recognizable speech and thus an active grammar may continue to receive result events.

Well-behaved applications adhere to the following conventions to minimize impact on other applications and other components of the same application:

- Never apply the `GLOBAL` activation mode to a `DictationGrammar`.
- Always use the default activation mode `RECOGNIZER_FOCUS` unless there is a good reason to use another mode.
- Only use the `RECOGNIZER_MODAL` when certain that deactivating `RECOGNIZER_FOCUS` grammars will not adversely affect the user interface.
- Minimize the complexity and the number of `RuleGrammars` with `GLOBAL` activation mode.
- 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.

The general principal underlying these conventions is that increasing the number of active grammars and/or increasing the complexity of those grammars can lead to slower response time, greater CPU load and reduced recognition accuracy (more mistakes).

Committing Changes

Grammars can be dynamically changed and enabled and disabled. Changes may be necessary as the application changes context, as new information becomes available and so on. As with graphical interface programming most grammar updates occur during processing of events. Very often grammars are updated in response to speech input from a user (a `ResultEvent`). Other asynchronous events (e.g., `AWTEvents`) are another common trigger of grammar changes. Changing grammars during normal operation of a recognizer is usually necessary to ensure natural and usable speech-enabled applications.

Different grammar types allow different types of run-time change. `RuleGrammars` can be created and deleted during normal recognizer operation. Also, any aspect of a created `RuleGrammar` can be redefined: rules can be modified, deleted or added, imports can be changed an so on. Certain properties of a `DictationGrammar` can be changed: the context can be updated and the vocabulary modified.

For any grammar changes to take effect they must be *committed*. Committing changes builds the grammars into a format that can be used by the internal processes of the recognizer and applies those changes to the processing of incoming audio.

There are two ways in which a commit takes place:

1. An explicit call by an application to the `commitChanges` method of the `Recognizer`. The documentation for the method describes when and how changes are committed when called. (The `commitChanges` method is typically used in conjunction with the `suspend` method of `Recognizer`.)
2. Changes to all grammars are implicitly committed at the completion of processing of a result finalization event (either a `RESULT_ACCEPTED` or `RESULT_REJECTED` event). This simplifies the common scenario in which grammars are changed during result finalization process because the user's input has changed the application's state. (This implicit commit is deferred following a call to `suspend` and until a call to `commitChanges`.)

The `Recognizer` issues a `CHANGES_COMMITTED` event to signal that changes have been committed. This event signals a transition from the `SUSPENDED` state to the `LISTENING` state. Once in the `LISTENING` state the `Recognizer` resumes the processing of incoming audio with the newly committed grammars.

Also each changed `Grammar` receives a `GRAMMAR_CHANGES_COMMITTED` through attached `GrammarListeners` whenever changes to it are committed.

The commit changes mechanism has two important properties:

- Updates to grammar definations 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.
- It 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.

Grammar and Result Listeners

An application can attach one or more `GrammarListeners` to any `Grammar`. The listener is notified of status changes for the `Grammar`: when changes are committed and when activation changes.

An application can attach one or more `ResultListener` objects to each `Grammar`. The listener is notified of all events for all results that match this grammar. The listeners receive `ResultEvents` starting with the `GRAMMAR_FINALIZED` event (not the preceding `RESULT_CREATED` or `RESULT_UPDATED` events).

See Also:
RuleGrammar, DictationGrammar, GrammarListener, ResultListener, Result, Recognizer, commitChanges

---

Field Summary

static int	**GLOBAL** Value of activation mode in which the `Grammar` is active for recognition irrespective of the focus state of the `Recognizer`.
static int	**RECOGNIZER_FOCUS** Default value of activation mode that requires the `Recognizer` have focus and that there be no enabled grammars with `RECOGNIZER_MODAL` mode for the grammar to be activated.
static int	**RECOGNIZER_MODAL** Value of activation mode that requires the `Recognizer` have focus for the grammar to be activated.

Method Summary

void	**addGrammarListener**(GrammarListener listener) Request notifications of events of related to this `Grammar`.
void	**addResultListener**(ResultListener listener) Request notifications of events from any `Result` that matches this `Grammar`.
int	**getActivationMode**() Return the current activation mode for a `Grammar`.
java.lang.String	**getName**() Get the name of a grammar.
Recognizer	**getRecognizer**() Returns a reference to the `Recognizer` that owns this grammar.
boolean	**isActive**() Test whether a `Grammar` is currently active for recognition.
boolean	**isEnabled**() Return the `enabled` property of a `Grammar`.
void	**removeGrammarListener**(GrammarListener listener) Remove a listener from this `Grammar`.
void	**removeResultListener**(ResultListener listener) Remove a `ResultListener` from this `Grammar`.
void	**setActivationMode**(int mode) Set the activation mode of a `Grammar` as `RECOGNIZER_FOCUS`, `RECOGNIZER_MODAL`, or `GLOBAL`.
void	**setEnabled**(boolean enabled) Set the enabled property of a `Grammar`.

Field Detail

RECOGNIZER_FOCUS

`public static final int` **`RECOGNIZER_FOCUS`**

Default value of activation mode that requires the `Recognizer` have focus and that there be no enabled grammars with `RECOGNIZER_MODAL` mode for the grammar to be activated. This is the lowest priority activation mode and should be used unless there is a user interface design reason to use another mode.

The activation conditions for the `RECOGNIZER_FOCUS` mode are describe above.

See Also:
setActivationMode, RECOGNIZER_MODAL, GLOBAL

RECOGNIZER_MODAL

```
public static final int RECOGNIZER_MODAL
```

Value of activation mode that requires the `Recognizer` have focus for the grammar to be activated.

The activation conditions for the `RECOGNIZER_MODAL` mode are describe above.

See Also:
setActivationMode, RECOGNIZER_FOCUS, GLOBAL

GLOBAL

```
public static final int GLOBAL
```

Value of activation mode in which the `Grammar` is active for recognition irrespective of the focus state of the `Recognizer`.

The activation conditions for the `GLOBAL` mode are describe above.

See Also:
setActivationMode, RECOGNIZER_FOCUS, RECOGNIZER_MODAL

Method Detail

getRecognizer

```
public Recognizer getRecognizer()
```

Returns a reference to the `Recognizer` that owns this grammar.

---

getName

```
public java.lang.String getName()
```

Get the name of a grammar. A grammar's name must be unique for a recognizer. Grammar names use a similar naming convention to Java classes. The naming convention are defined in the Java Speech Grammar Format Specification.

Grammar names are used with a `RuleGrammar` for resolving imports and references between grammars. The name of a `RuleGrammar` is set when the grammar is created (either by loading a JSGF document or creating a new `RuleGrammar`).

The name of a `DictationGrammar` should reflect the language domain it supports. For example: `com.acme.dictation.us.general` for general US Dictation from Acme corporation. Since a `DictationGrammar` is built into a `Recognizer`, its name is determined by the `Recognizer`.

See Also:
DictationGrammar, RuleGrammar

---

setEnabled

```
public void setEnabled(boolean enabled)
```

Set the enabled property of a `Grammar`. A change in the enabled property takes effect only after grammar changes are committed. Once a grammar is enabled and when the activation conditions are met, it is activated for recognition. When a grammar is activated, the `Recognizer` listens to incoming audio for speech that matches the grammar and produces a `Result` when matching speech is detected.

The enabled property of a grammar is tested with the `isEnabled` method. The activation state of a grammar is tested with the `isActive` method.

The `RuleGrammar` interface extends the enabling property to allow individual rules to be enabled and disabled.

See Also:
commitChanges, setEnabled(boolean), setEnabled(java.lang.String, boolean), setEnabled(java.lang.String[], boolean), isEnabled

---

isEnabled

```
public boolean isEnabled()
```

Return the `enabled` property of a `Grammar`. More specialized behaviour is specified by the `RuleGrammar` interface.

See Also:
setEnabled, isEnabled(), isEnabled(java.lang.String)

---

setActivationMode

```
public void setActivationMode(int mode)
                       throws java.lang.IllegalArgumentException
```

Set the activation mode of a `Grammar` as `RECOGNIZER_FOCUS`, `RECOGNIZER_MODAL`, or `GLOBAL`. The role of the activation mode in the activation conditions for a `Grammar` are described above. The default activation mode - `RECOGNIZER_FOCUS` - should be used unless there is a user interface design reason to use another mode.

The individual rules of a `RuleGrammar` can be separately enabled and disabled. However, all rules share the same `ActivationMode` since the mode is a property of the complete `Grammar`. A consequence is that all enabled rules of a `RuleGrammar` are activated and deactivated together.

A change in activation mode only takes effect once changes are committed. For some recognizers changing the activation mode is computationally expensive.

The activation mode of a grammar can be tested by the `getActivationMode` method.

[Note: future releases may modify the set of activation modes.]

Throws:
java.lang.IllegalArgumentException - if an attempt is made to set `GLOBAL` mode for a `DictationGrammar`

See Also:
getActivationMode, RECOGNIZER_FOCUS, RECOGNIZER_MODAL, GLOBAL, setEnabled, isActive, commitChanges

---

getActivationMode

`public int` **`getActivationMode`**`()`

Return the current activation mode for a `Grammar`. The default value for a grammar is `RECOGNIZER_FOCUS`.

See Also:
setActivationMode, setEnabled, isActive

---

isActive

`public boolean` **`isActive`**`()`

Test whether a `Grammar` is currently active for recognition. When a grammar is active, the recognizer is matching incoming audio against the grammar (and other active grammars) to detect speech that matches the grammar.

A `Grammar` is activated for recognition if the `enabled` property is set to `true` and the activation conditions are met. Activation is not directly controlled by applications and so can only be tested (there is no `setActive` method).

Rules of a `RuleGrammar` can be individuallly enabled and disabled. However all rules share the same `ActivationMode` and the same activation state. Thus, when a `RuleGrammar` is active, all the enabled rules of the grammar are active for recognition.

Changes in the activation state are indicated by `GRAMMAR_ACTIVATED` and `GRAMMAR_DEACTIVATED` events issued to the `GrammarListener`. A change in activation state can follow these `RecognizerEvents`:

- A `CHANGES_COMMITTED` event that applies a change in the `enabled` state or `ActivationMode` of this or another `Grammar`.
- A `FOCUS_GAINED` or `FOCUS_LOST` event.

See Also:
setEnabled, setActivationMode, GRAMMAR_ACTIVATED, GRAMMAR_DEACTIVATED, CHANGES_COMMITTED, FOCUS_GAINED, FOCUS_LOST

---

addGrammarListener

`public void` **`addGrammarListener`**`(GrammarListener listener)`

Request notifications of events of related to this `Grammar`. An application can attach multiple listeners to a `Grammar`.

See Also:
removeGrammarListener

---

removeGrammarListener

`public void` **`removeGrammarListener`**`(GrammarListener listener)`

Remove a listener from this `Grammar`.

See Also:
addGrammarListener

---

addResultListener

`public void` **`addResultListener`**`(ResultListener listener)`

Request notifications of events from any `Result` that matches this `Grammar`. An application can attach multiple ResultListeners to a `Grammar`. A listener is removed with the `removeResultListener` method.

A `ResultListener` attached to a `Grammar` receives result events starting from the `GRAMMAR_FINALIZED` event - the event which indicates that the matched grammar is known. A `ResultListener` attached to a `Grammar` will never receive a `RESULT_CREATED` event and does not receive any `RESULT_UPDATED` events that occurred before the `GRAMMAR_FINALIZED` event. A `ResultListener` attached to a `Grammar` is guaranteed to receive a result finalization event - `RESULT_ACCEPTED` or `RESULT_REJECTED` - some time after the `GRAMMAR_FINALIZED` event.

`ResultListener` objects can also be attached to a `Recognizer` and to any `Result`. A listener attached to the `Recognizer` receives all events for all results produced by that `Recognizer`. A listener attached to a `Result` receives all events for that result from the time at which the listener is attached.

See Also:
removeResultListener, addResultListener, addResultListener, RESULT_CREATED, GRAMMAR_FINALIZED, RESULT_ACCEPTED, RESULT_REJECTED

---

removeResultListener

`public void` **`removeResultListener`**`(ResultListener listener)`

Remove a `ResultListener` from this `Grammar`.

See Also:
addResultListener, removeResultListener, removeResultListener

---

Class javax.speech.recognition.GrammarAdapter

```
java.lang.Object
  |
  +--javax.speech.recognition.GrammarAdapter
```

public class **GrammarAdapter**
extends java.lang.Object
implements GrammarListener

The adapter which receives grammar events. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**grammarActivated**(GrammarEvent e) A `GRAMMAR_ACTIVATED` event occurred.
void	**grammarChangesCommitted**(GrammarEvent e) Event issued when a `Recognizer` has committed changes to a `Grammar`.
void	**grammarDeactivated**(GrammarEvent e) A `GRAMMAR_DEACTIVATED` event occurred.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

grammarChangesCommitted

`public void grammarChangesCommitted(GrammarEvent e)`

Event issued when a `Recognizer` has committed changes to a `Grammar`. The `Grammar` interface documents how and when changes are committed.

Specified by:
grammarChangesCommitted in interface GrammarListener
See Also:
GRAMMAR_CHANGES_COMMITTED

grammarActivated

`public void grammarActivated(GrammarEvent e)`

A `GRAMMAR_ACTIVATED` event occurred.

Specified by:
grammarActivated in interface GrammarListener
See Also:
GRAMMAR_ACTIVATED

grammarDeactivated

`public void grammarDeactivated(GrammarEvent e)`

A `GRAMMAR_DEACTIVATED` event occurred.

Specified by:
grammarDeactivated in interface GrammarListener
See Also:
GRAMMAR_DEACTIVATED

Class javax.speech.recognition.GrammarEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.recognition.GrammarEvent
```

public class **GrammarEvent**
extends SpeechEvent

A `GrammarEvent` is issued to each `GrammarListener` attached to a `Grammar` when major events associated with that `Grammar` occur.

The source for a `GrammarEvent` is always a `Grammar` object.

See Also:
Grammar, GrammarListener, Serialized Form

Field Summary

boolean	**definitionChanged** True if the grammar's definition has changed with a `GRAMMAR_CHANGES_COMMITTED` event.
boolean	**enabledChanged** True if the grammar's `enabled` property has changed with a `GRAMMAR_CHANGES_COMMITTED` event.
static int	**GRAMMAR_ACTIVATED** A `GRAMMAR_ACTIVATED` event is issued when a grammar changes state from deactivated to activated.
static int	**GRAMMAR_CHANGES_COMMITTED** A `GRAMMAR_CHANGES_COMMITTED` event is issued when a `Recognizer` completes committing changes to a `Grammar`.
static int	**GRAMMAR_DEACTIVATED** A `GRAMMAR_DEACTIVATED` event is issued when a grammar changes state from activated to deactivated.
GrammarException	**grammarException** Non-null if any error is detected in a grammar's definition while producing a `GRAMMAR_CHANGES_COMMITTED` event.

Fields inherited from class javax.speech.SpeechEvent	
id	

Fields inherited from class java.util.EventObject
source

Constructor Summary
GrammarEvent(Grammar source, int id, boolean enabledChanged, boolean definitionChanged, GrammarException grammarException) Constructs a `GrammarEvent` event with a specified event identifier plus state change and exception values.
GrammarEvent(Grammar source, int id) Constructs a `GrammarEvent` event with a specified event identifier.

Method Summary	
boolean	**getDefinitionChanged**() Returns `true` for a `GRAMMAR_CHANGES_COMMITTED` event if the definition of the source `Grammar` has changed.
boolean	**getEnabledChanged**() Returns `true` for a `GRAMMAR_CHANGES_COMMITTED` event if the `enabled` property of the `Grammar` changed.
GrammarException	**getGrammarException**() Returns non-null for a `GRAMMAR_CHANGES_COMMITTED` event if an error is found in the grammar definition.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

GRAMMAR_CHANGES_COMMITTED

`public static final int GRAMMAR_CHANGES_COMMITTED`

A `GRAMMAR_CHANGES_COMMITTED` event is issued when a `Recognizer` completes committing changes to a `Grammar`. The event is issued immediately following the `CHANGES_COMMITTED` event that is issued to `RecognizerListeners`. That event indicates that changes have been applied to all grammars of a `Recognizer`. The `GRAMMAR_CHANGES_COMMITTED` event is specific to each individual grammar.

The event is issued when the definition of the grammar is changed, when its `enabled` property is changed, or both. The `enabledChanged` and `definitionChanged` flags are set accordingly.

A `GRAMMAR_CHANGES_COMMITTED` event can triggered without an explicit call to `commitChanges` - there is usually an implicit `commitChanges` at the completion of result finalization event processing. If any syntactic or logical errors are detected for a `Grammar` during the commit, the generated `GrammarException` is included with this event. If no problem is found the value is `null`.

See Also:
grammarChangesCommitted, CHANGES_COMMITTED, commitChanges, getDefinitionChanged, getEnabledChanged, getGrammarException

---

GRAMMAR_ACTIVATED

`public static final int GRAMMAR_ACTIVATED`

A `GRAMMAR_ACTIVATED` event is issued when a grammar changes state from deactivated to activated. The `isActive` method of the `Grammar` will now return `true`.

Grammar activation changes follow one of two `RecognizerEvents`: (1) a `CHANGES_COMMITTED` event in which a grammar's `enabled` flag is set `true` or (2) a `FOCUS_GAINED` event. The full details of the activation conditions under which a `Grammar` is activated are described in the documentation for the `Grammar` interface.

See Also:
grammarActivated, CHANGES_COMMITTED, FOCUS_GAINED

GRAMMAR_DEACTIVATED

```
public static final int GRAMMAR_DEACTIVATED
```

A `GRAMMAR_DEACTIVATED` event is issued when a grammar changes state from activated to deactivated. The `isActive` method of the `Grammar` will now return `false`.

Grammar deactivation changes follow one of two `RecognizerEvents`: (1) a `CHANGES_COMMITTED` event in which a grammar's `enabled` flag is set `false` or (2) a `FOCUS_LOST` event. The full details of the activation conditions under which a `Grammar` is deactivated are described in the documentation for the `Grammar` interface.

See Also:
grammarDeactivated, CHANGES_COMMITTED, FOCUS_LOST

---

enabledChanged

```
protected boolean enabledChanged
```

True if the grammar's `enabled` property has changed with a `GRAMMAR_CHANGES_COMMITTED` event. False for other event types.

See Also:
getEnabledChanged

---

definitionChanged

```
protected boolean definitionChanged
```

True if the grammar's definition has changed with a `GRAMMAR_CHANGES_COMMITTED` event. False for other event types.

See Also:
getDefinitionChanged

---

grammarException

```
protected GrammarException grammarException
```

Non-null if any error is detected in a grammar's definition while producing a `GRAMMAR_CHANGES_COMMITTED` event. `null` for other event types.

See Also:
getGrammarException

Constructor Detail

GrammarEvent

```
public GrammarEvent(Grammar source,
                    int id,
                    boolean enabledChanged,
                    boolean definitionChanged,
                    GrammarException grammarException)
```

Constructs a `GrammarEvent` event with a specified event identifier plus state change and exception values. For a `GRAMMAR_CHANGES_COMMITTED` event, the `enabledChanged` and `definitionChanged` parameters should indicate what properties of the `Grammar` has changed, otherwise they should be `false`. For a `GRAMMAR_CHANGES_COMMITTED` event, the `grammarException` parameter should be non-null only if an error is encountered in the grammar definition.

Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type
`enabledChanged` - `true` if the grammar's `enabled` property changed
`definitionChanged` - `true` if the grammar's definition has changed
`grammarException` - non-null if an error is detected in a grammar's definition

---

GrammarEvent

```
public GrammarEvent(Grammar source,
                    int id)
```

Constructs a `GrammarEvent` event with a specified event identifier. The `enabledChanged` and `definitionChanged` fields are set to `false`. The `grammarException` field is set to `null`.

Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type

Method Detail

getEnabledChanged

```
public boolean getEnabledChanged()
```

Returns `true` for a `GRAMMAR_CHANGES_COMMITTED` event if the `enabled` property of the `Grammar` changed.

---

getDefinitionChanged

```
public boolean getDefinitionChanged()
```

Returns `true` for a `GRAMMAR_CHANGES_COMMITTED` event if the definition of the source `Grammar` has changed.

getGrammarException

`public GrammarException` **`getGrammarException`**`()`

Returns non-null for a `GRAMMAR_CHANGES_COMMITTED` event if an error is found in the grammar definition.

paramString

`public java.lang.String` **`paramString`**`()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event

Overrides:
paramString in class SpeechEvent

Class javax.speech.recognition.GrammarException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--javax.speech.SpeechException
                    |
                    +--javax.speech.recognition.GrammarException
```

public class **GrammarException**
extends SpeechException

Thrown if a problem is found with a Java Speech Grammar Format (JSGF) file or with a `RuleGrammar` object derived from JSGF.

Grammar problems are typically identified and fixed during application development. This class provides information that allows a debugging environment to handle the error.

The exception message is a printable string. Recognizers may optionally provide details of each syntax problem.

See Also:
Serialized Form

Constructor Summary
GrammarException() Constructs a `GrammarException` with no detail message.
GrammarException(java.lang.String s) Constructs a `GrammarException` with the specified detail message.
GrammarException(java.lang.String s, GrammarSyntaxDetail[] details) Constructs a `GrammarException` with the specified detail message and an optional programmatic description of each error.

Method Summary	
void	**addDetail**(GrammarSyntaxDetail detail) Add a syntax error description (appended to the existing array of details).
GrammarSyntaxDetail[]	**getDetails**() Return the list of grammar syntax problem descriptions.
void	**setDetails**(GrammarSyntaxDetail[] details) Set the grammar syntax problem descriptions.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

GrammarException

```
public GrammarException()
```

Constructs a `GrammarException` with no detail message.

GrammarException

```
public GrammarException(java.lang.String s)
```

Constructs a `GrammarException` with the specified detail message.

Parameters:
`s` - a printable detail message

GrammarException

```
public GrammarException(java.lang.String s,
                        GrammarSyntaxDetail[] details)
```

Constructs a `GrammarException` with the specified detail message and an optional programmatic description of each error.

Parameters:
`s` - a printable detail message
`detail` - detail of each error encountered or `null`

Method Detail

getDetails

```
public GrammarSyntaxDetail[] getDetails()
```

Return the list of grammar syntax problem descriptions.

---

setDetails

```
public void setDetails(GrammarSyntaxDetail[] details)
```

Set the grammar syntax problem descriptions.

---

addDetail

```
public void addDetail(GrammarSyntaxDetail detail)
```

Add a syntax error description (appended to the existing array of details).

Interface javax.speech.recognition.GrammarListener

Implementing Classes:
GrammarAdapter

---

public abstract interface **GrammarListener**
extends java.util.EventListener

A `GrammarListener` receives notifications of status change events for a `Grammar`.

A `GrammarListener` is attached to and removed from a `Grammar` with its `addGrammarListener` and `removeGrammarListener` methods. Multiple grammars can share a `GrammarListener` object and one grammar can have multiple `GrammarListener` objects attached.

The `GrammarAdapter` provides a trivial implementation of this class.

See Also:
Grammar, DictationGrammar, RuleGrammar, addGrammarListener, removeGrammarListener, GrammarAdapter

---

Method Summary

void	**grammarActivated**(GrammarEvent e) A `GRAMMAR_ACTIVATED` event is issued when a grammar changes from deactivated to activated.
void	**grammarChangesCommitted**(GrammarEvent e) A `GRAMMAR_CHANGES_COMMITTED` event is issued when a `Recognizer` has committed changes to a `Grammar`.
void	**grammarDeactivated**(GrammarEvent e) A `GRAMMAR_DEACTIVATED` event is issued when a grammar changes from activated to deactivated.

Method Detail

grammarChangesCommitted

```
public void grammarChangesCommitted(GrammarEvent e)
```

A `GRAMMAR_CHANGES_COMMITTED` event is issued when a `Recognizer` has committed changes to a `Grammar`.

The `GRAMMAR_CHANGES_COMMITTED` immediately follows the `CHANGES_COMMITTED` event issued to `RecognizerListeners` after changes to all grammars have been committed. The circumstances in which changes are committed are described in the documentation for `Grammar`.

See Also:
GRAMMAR_CHANGES_COMMITTED, CHANGES_COMMITTED, Grammar

---

grammarActivated

```
public void grammarActivated(GrammarEvent e)
```

A `GRAMMAR_ACTIVATED` event is issued when a grammar changes from deactivated to activated.

See Also:
GRAMMAR_ACTIVATED, Grammar

---

grammarDeactivated

```
public void grammarDeactivated(GrammarEvent e)
```

A `GRAMMAR_DEACTIVATED` event is issued when a grammar changes from activated to deactivated.

See Also:
GRAMMAR_DEACTIVATED, Grammar

Class javax.speech.recognition.GrammarSyntaxDetail

```
java.lang.Object
  |
  +--javax.speech.recognition.GrammarSyntaxDetail
```

---

public class **GrammarSyntaxDetail**
extends java.lang.Object

Description of a problem found in a grammar usually bundled with a `GrammarException`. The grammar may have been created programmatically or by loading a Java Speech Grammar Format document. Multiple `GrammarSyntaxDetail` objects may be encapsulated by a single `GrammarException`.

Depending on the type of error and the context in which the error is identified, some or all of the following information may be provided:

- Grammar name,
- Grammar location (URL or system resource),
- Rule name in which error is found,
- Import name,
- Line number of error in JSGF,
- Character number (within line) in JSGF,
- A printable description string.

The following problems may be encountered when loading JSGF from a `URL` or `Reader`, or through the `ruleForJSGF` method of the `RuleGrammar` interface.

- Missing or illegal grammar name declaration, or grammar name doesn't match URL or file name.
- missing URL or URL does not contain JSGF file.
- Illegal import declaration.
- Illegal rule name or token.
- Missing semi-colon.
- Redefinition of a rule.
- Definition of a reserved rule name (`<NULL>`, `<VOID>`).
- Unclosed quotes, tags, comment, or rule name.
- Unclosed grouping "()" or "[]".
- Empty rule definition or empty alternative.
- Missing or illegal weight on alternatives.
- Illegal attachment of unary operators (count and tag).
- Some other error.

When the `commitChanges` method of a `Recognizer` is called, it performs addition checks to ensure all loaded grammars are legal. The following problems may be encountered:

- Unable to resolve import because grammar or rule is not defined.
- Reference to an undefined rule name.
- Illegal recursion: a rule refers to itself illegally. Only right recursion is allowed (defined by JSGF).

- Ambiguous reference to imported rule.

See Also:
GrammarException, loadJSGF(java.io.Reader), loadJSGF(java.net.URL, java.lang.String), ruleForJSGF, commitChanges

---

Field Summary

int	**charNumber** Character number in line in JSGF file for problem.
java.net.URL	**grammarLocation** URL location of grammar in which problem is encountered.
java.lang.String	**grammarName** Name of grammar in which problem is encountered.
RuleName	**importName** Name in grammar import declaration in which problem is encountered.
int	**lineNumber** Line number in JSGF file for problem.
java.lang.String	**message** Printable string describing problem.
java.lang.String	**ruleName** Name of rule within grammar in which problem is encountered.

Constructor Summary

GrammarSyntaxDetail() Empty constructor.
GrammarSyntaxDetail(java.lang.String grammarName, java.net.URL grammarLocation, java.lang.String ruleName, RuleName importName, int lineNumber, int charNumber, java.lang.String message) Complete constructor describing a syntax problem.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

grammarName

```
public java.lang.String grammarName
```

Name of grammar in which problem is encountered. May be `null`.

---

grammarLocation

```
public java.net.URL grammarLocation
```

URL location of grammar in which problem is encountered. May be `null`.

---

ruleName

```
public java.lang.String ruleName
```

Name of rule within grammar in which problem is encountered. May be `null`.

---

importName

```
public RuleName importName
```

Name in grammar import declaration in which problem is encountered. May be `null`.

---

lineNumber

```
public int lineNumber
```

Line number in JSGF file for problem. Negative values indicate that the line number unknown.

---

charNumber

```
public int charNumber
```

Character number in line in JSGF file for problem. Negative values indicate that the line number unknown.

---

message

```
public java.lang.String message
```

Printable string describing problem. May be `null`.

Constructor Detail

GrammarSyntaxDetail

```
public GrammarSyntaxDetail()
```

Empty constructor.

GrammarSyntaxDetail

```
public GrammarSyntaxDetail(java.lang.String grammarName,
                           java.net.URL grammarLocation,
                           java.lang.String ruleName,
                           RuleName importName,
                           int lineNumber,
                           int charNumber,
                           java.lang.String message)
```

Complete constructor describing a syntax problem.

Interface javax.speech.recognition.Recognizer

public abstract interface **Recognizer**
extends Engine

A `Recognizer` provides access to speech recognition capabilities. The `Recognizer` interface extends the `javax.speech.Engine` interface and so inherits the basic engine capabilities and provides additional specialized capabilities.

The primary capabilities provided by a recognizer are grammar management and result handling. An application is responsible for providing a recognizer with grammars. A `Grammar` defines a set of words (technically known as tokens) and defines patterns in which the tokens may be spoken. When a grammar is *active* the recognizer listens for speech in the incoming audio which matches the grammar. When speech is detected the recognizer produces a `Result`. The result object is passed to the application and contains information on which words were heard.

The types of grammars, mechanisms for creating, modifying and managing grammars, types of results, result handling and other recognizer functions are described in more detail below.

Creating a Recognizer

A `Recognizer` is created by a call to `createRecognizer` method of the `Central` class. Detailed descriptions of the procedures for locating, selecting, creating and initializing a Recognizer are provided in the documentation for `Central`.

Inherited and Extended Engine Capabilities

The `Recognizer` interface and the other classes and interfaces of the `javax.speech.recognition` package extend and modify the basic engine capabilities in the following ways.

- Inherits engine location by `Central.availableRecognizers` method and `EngineModeDesc`.
- Extends `EngineModeDesc` as `RecognizerModeDesc`.
- Inherits `allocate` and `deallocate` methods from `Engine`.
- Inherits `pause` and `resume` methods from `Engine`.
- Inherits the `getEngineState`, `waitEngineState` and `testEngineState` methods from the `Engine` interface for handling engine state.
- Inherits the `DEALLOCATED`, `ALLOCATED`, `ALLOCATING_RESOURCES` and `DEALLOCATING_RESOURCES` states from the `Engine` interface.
- Inherits the `PAUSED` and `RESUMED` sub-states of the `ALLOCATED` state.
- Adds `PROCESSING`, `LISTENING` and `SUSPENDED` sub-states of the `ALLOCATED` state.
- Adds `FOCUS_ON` and `FOCUS_OFF` sub-states of the `ALLOCATED` state.
- Adds `requestFocus` and `releaseFocus` methods for managing an application's speech focus.
- Adds special suspend and commit mechanisms for updating definitions and enabling of grammars: `suspend`, `commitChanges`.
- Inherits audio management: see `Engine.getAudioManager` and `AudioManager`.
- Inherits the `Engine.getEngineProperties` method adds the `getRecognizerProperties` so that a cast is not required.
- Extends audio event mechanism: see `RecognizerAudioListener` and `RecognizerAudioEvent`.
- Inherits vocabulary management: see `Engine.getVocabManager` and `VocabManager`.
- Inherits `addEngineListener` and `removeEngineListener` methods from the `Engine` interface.
- Extends `EngineListener` interface as `RecognizerListener`.
- Adds grammar management of `Grammar`, `RuleGrammar`, `DictationGrammar` `GrammarEvent`, and `GrammarListener`,
- Adds recognition result mechanisms through `Result`, `FinalResult`, `FinalRuleResult`, `FinalDictationResult`, `ResultListener` and `ResultEvent`.

Grammars

The creation, modification, enabling, activation and other management of grammars is a core function provided by any recognizer. A recognizer must be provided with one or more grammars that indicate what words and word sequences it should listen for. The basic process for dealing with a grammar are:

1. Create a new `Grammar` or obtain a reference to an existing grammar.
2. Attach a `ResultListener` to either the `Recognizer` or `Grammar` to get `Result` events.
3. As necessary, setup or modify the grammar according to the application context.
4. Enable and disable the `Grammar` for recognition as required.
5. Commit changes to grammar definition and enabled status.
6. Repeat steps 1 through 5 as required.
7. Delete application-created grammars when they are no longer needed.

Each grammar must be identified by a unique grammar name which is defined when the `Grammar` is created. The following methods deal with grammar names:

- `Grammar.getName` returns the name of a `Grammar` object.
- `getRuleGrammar` returns a reference to a Grammar given the grammar's names.

A `Recognizer` provides several methods for the creation and loading of `RuleGrammars`:

- `newRuleGrammar`: create a `RuleGrammar` from scratch
- `loadJSGF`: create a `RuleGrammar(s)` from Java Speech Grammar Format text obtained from either a `Reader` or a `URL`. The advanced `loadJSGF` method provides additional load controls.
- `readVendorGrammar`: read a grammar stored in a vendor-specific (non-portable) format.

Other important rule grammar functions are:

- `deleteRuleGrammar` deletes a loaded grammar.
- `listRuleGrammars` returns an array of references to all grammars loaded into a `Recognizer`.
- `RuleGrammar.toString` produces a string representing a `RuleGrammar` in Java Speech Grammar Format.
- `writeVendorGrammar`: write a grammar in a vendor-specific (non-portable) format.

In addition to `RuleGrammars` that an application creates, some recognizers have "built-in" grammars. A built-in grammar is automatically loaded into the recognizer when the recognizer is created and are accessible through the `listRuleGrammars` method. Different recognizers will have different built-in grammars so applications should not rely upon built-in grammars if they need to be portable.

A `DictationGrammar` is the most important kind of built-in grammar. A dictation grammar supports relatively free-form input of text. If a recognizer supports dictation, then the `getDictationGrammar` method returns non-null and the `isDictationGrammarSupported` method of the `RecognizerModeDesc` returns true.

Grammar Scope

Each `Recognizer` object has a separate name-space. Applications should be aware that changes to a `Grammar` object affect any part of the application that uses that `Grammar`. However, if two separate applications create a `Recognizer` and use a `Grammar` with the same name, they are effectively independent - changes by on app do not affect the operation of the other app.

Efficiency

The processing of grammars (particularly large grammars) can be computationally expensive. Loading a new grammar can take seconds or sometimes minutes. Updates to a `Grammar` may also be slow. Therefore, applications should take precautions to build grammars in advance and to modify them only when necessary. Furthermore, an application should minimize the number of grammars active at any point in time.

Recognizer States

A `Recognizer` inherits the `PAUSED` and `RESUMED` sub-states of the `ALLOCATED` from the `Engine` interface. The `Recognizer` interface adds two more independent sub-states systems to the `ALLOCATED` state:

- `LISTENING`, `PROCESSING`, `SUSPENDED`
- `FOCUS_ON`, `FOCUS_OFF`

Recognizer States: Focus

A `Recognizer` adds the two state sub-state-system for `FOCUS_ON` and `FOCUS_OFF` to indicate whether this instance of the `Recognizer` currently has the speech focus. Focus is important in a multi-application environment because more than one application can connect to an underlying speech recognition engine, but the user gives speech focus to only one application at a time. Since it is normal for an application to use only one `Recognizer`, `Recognizer` focus and application focus normally mean the same thing. (Multi-recognizer apps cannot make this assumption.)

Focus is not usually relevant in telephony applications in which there is a single input audio stream to a single application.

The focus status is a key factor in determining the activation of grammars and therefore in determining when results will and will not be generated. The activation conditions for grammars and the role of focus are described in the documentation for the `Grammar` interface.

When recognizer focus is received a `FOCUS_GAINED` event is issued to `RecognizerListeners`. When recognizer focus is released or otherwise lost, a `FOCUS_LOST` event is issued to `RecognizerListeners`.

Applications can request speech focus for the `Recognizer` by calling the `requestFocus` mechanism. This asynchronous method may return before focus in received. To determine when focus is on, check for `FOCUS_GAINED` events or test the `FOCUS_ON` bit of engine state.

Applications can release speech focus from the `Recognizer` by calling the `releaseFocus` mechanism. This asynchronous method may return before focus in lost. To determine whether focus is off, check for `FOCUS_LOST` events or test the `FOCUS_OFF` bit of engine state.

In desktop environments, it is normal (but not a requirement) for speech focus to follow window focus. Therefore, it is common for the `requestFocus` method to be called on AWT events or Swing events such as `FocusEvent` and `WindowEvent`.

A well-behaved application only requests focus when it knows that it has the speech focus of the user (the user is talking to it and not to other applications). A well-behaved application will also release focus as soon as it finishes with it.

Recognizer States: Result Recognition

A `Recognizer` adds the three state sub-state-system for `LISTENING`, `PROCESSING` and `SUSPENDED` to indicate the status of the recognition of incoming speech. These three states are loosely coupled with the `PAUSED` and `RESUMED` states but only to the extent that turning on and off audio input will affect the recognition process. An `ALLOCATED` recognizer is always in one of these 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.
- `PROCESSING` state: the `Recognizer` is processing incoming speech that may match an active grammar to produce 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.

Recognizer States: Result Recognition: Typical Event Cycle

The typical state cycle is as follows.

1. A `Recognizer` starts in the `LISTENING` state with a certain set of grammars enabled. When incoming audio is detected that may match an active grammar, the `Recognizer` transitions to the `PROCESSING` state with a `RECOGNIZER_PROCESSING` `RecognizerEvent`. The `Recognizer` then creates a new `Result` and issues a `RESULT_CREATED` event to provide it to the application.
2. The `Recognizer` remains in the `PROCESSING` state until it completes recognition of the result.
3. The recognizer indicates completion of recognition by issuing a `RECOGNIZER_SUSPENDED` `RecognizerEvent` to transition from the `PROCESSING` state to the `SUSPENDED` state. Once in that state, it issues a result finalization event to `ResultListeners` (`RESULT_ACCEPTED` or `RESULT_REJECTED` event).
4. The `Recognizer` remains in the `SUSPENDED` state until processing of the result finalization event is completed. Applications will usually make any necessary grammar changes while the recognizer is `SUSPENDED`. In this 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.

5. The `Recognizer` commits all grammar changes, issues a `CHANGES_COMMITTED` event to `RecognizerListeners` 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, typically including changes made in the result finalization events.
6. The `Recognizer` is back in the `LISTENING` state listening for speech that matches the new grammars.

In this state cycle, the `RECOGNIZER_PROCESSING` and `RECOGNIZER_SUSPENDED` events are triggered by user actions: starting and stopping speaking. The third state transition -- `CHANGES_COMMITTED` -- is triggered programmatically some time after the `RECOGNIZER_SUSPENDED` event.

Recognizer States: Result Recognition: Non-Speech Events

For applications that deal only with spoken input the state cycle 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., `AWTEvents`), timer events, multi-threading events, socket events and much more.

When a non-speech event occurs which changes the application state or applicatin data it is often necessary to update the recognizer's grammars. Furthermore, it is typically necessary to do this as if the change occurred in real time - at exactly the point in time at which the event occurred.

The `suspend` and `commitChanges` methods of a `Recognizer` are used to handle non-speech asynchronous events. The typical cycle for updating grammars is as follows:

1. 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.
2. The recognizer issues a `RECOGNIZER_SUSPENDED` event and transitions from the `LISTENING` state to the `SUSPENDED` state.
3. The application makes all necessary changes to the grammars.
4. Once all changes are completed the application calls the `commitChanges` method. The recognizer applies the new grammars, issues a `CHANGES_COMMITTED` event to transition from the `SUSPENDED` state back to the `LISTENING`, and issues `GRAMMAR_CHANGES_COMMITTED` events to all changed grammars.
5. The `Recognizer` resumes recognition of the buffered audio and then live audio with the new grammars.

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 makes changes and `commitChanges` as quickly as possible. This minimizes the possibility of a buffer overrun. It also reduces delays in recognizing speech and responding to the user.

Note: an application is not technically required to call `suspend` prior to calling `commitChanges`. If the `suspend` call is ommitted the `Recognizer` behaves as if `suspend` had been called immediately prior to calling `commitChanges`. An application that does not call `suspend` risks a commit occurring unexpectedly and leaving grammars in an inconsistent state.

Recognizer States: Result Recognition: Mixing Speech and Non-Speech Events

There is no guarantee that a speech and non-speech events will not be mixed. If a speech event occurs in the absence of non-speech events, the normal event cycle takes place. If a non-speech event occurs in the absence of any speech events, the non-speech event cycle takes place.

We need to consider two cases in which speech and non-speech events interact: (1) when a non-speech event occurs during the processing of a speech event, and (2) when a speech event occurs during the processing of a non-speech event.

1. Non-speech event occurs during processing of a speech event:
   Technically, this is the case in which a non-speech event is issued while the `Recognizer` is in either the `PROCESSING` state or the `SUSPENDED` state. In both cases the event processing for the non-speech is no different than normal. The non-speech event handler calls `suspend` to indicate it is about to change grammars, makes the grammar changes, and then calls `commitChanges` to apply the changes.
   The effect is that the `CHANGES_COMMITTED` event that would normally occur in the normal event cycle may be delayed until the `commitChanges` method is explicitly called and that the commit applies changes made in response to both the speech and non-speech events. If the `commitChanges` call for the non-speech event is made before the end of the result finalization event, there is no delay of the `CHANGES_COMMITTED` event.
2. Speech event occurs during processing of a non-speech event:
   This case is simpler. If the user starts speaking while a non-speech event is being processed, then the `Recognizer` is in the `SUSPENDED` state, that speech is buffered, and the speech event is actually delayed until the `Recognizer` returns to the `LISTENING` state. Once the `Recognizer` returns to the `LISTENING` state, the incoming speech is processed with the normal event cycle.

See Also:
RecognizerEvent, EngineListener, RecognizerListener, Grammar, RuleGrammar, DictationGrammar, Result, FinalResult

Field Summary

static long	**FOCUS_OFF** `FOCUS_OFF` is the bit of state that is set when an `ALLOCATED` `Recognizer` does not have the speech focus of the underlying speech recognition engine.
static long	**FOCUS_ON** `FOCUS_ON` is the bit of state that is set when an `ALLOCATED` `Recognizer` has the speech focus of the underlying speech recognition engine.
static long	**LISTENING** `LISTENING` is the bit of state that is set when an `ALLOCATED` `Recognizer` is listening to incoming audio for speech that may match an active grammar but has not yet detected speech.
static long	**PROCESSING** `PROCESSING` is the bit of state that is set when an `ALLOCATED` `Recognizer` is producing a `Result` for incoming speech that may match an active grammar.
static long	**SUSPENDED** `SUSPENDED` is the bit of state that is set when an `ALLOCATED` `Recognizer` is temporarily suspended while grammar definition and grammar enabled settings are updated.

Method Summary

void	**addResultListener**(ResultListener listener) Request notifications of all events for all `Result` produced by this `Recognizer`.
void	**commitChanges**() Commit changes to all loaded grammars and all changes of grammar enabling since the last `commitChanges`.
void	**deleteRuleGrammar**(RuleGrammar grammar) Delete a `RuleGrammar` from the `Recognizer`.
void	**forceFinalize**(boolean flush) If the `Recognizer` is in the `PROCESSING` state (producing a `Result`), force the `Recognizer` to immediately complete processing of that result by finalizing it.
DictationGrammar	**getDictationGrammar**(java.lang.String name) Return the `DictationGrammar` for a `Recognizer`.
RecognizerProperties	**getRecognizerProperties**() Return the `RecognizerProperties` object (a JavaBean).
RuleGrammar	**getRuleGrammar**(java.lang.String name) Get the `RuleGrammar` with the specified name.
SpeakerManager	**getSpeakerManager**() Return an object which provides management of the speakers of a `Recognizer`.
RuleGrammar[]	**listRuleGrammars**() List the `RuleGrammars` known to the `Recognizer`.
RuleGrammar	**loadJSGF**(java.io.Reader JSGF) Create a `RuleGrammar` from Java Speech Grammar Format text provided by the `Reader`.
RuleGrammar	**loadJSGF**(java.net.URL context, java.lang.String grammarName) Load a `RuleGrammar` and its imported grammars from Java Speech Grammar Format text from URLs or from system resources.
RuleGrammar	**loadJSGF**(java.net.URL context, java.lang.String grammarName, boolean loadImports, boolean reloadGrammars, java.util.Vector loadedGrammars) Load a `RuleGrammar` in Java Speech Grammar Format text from a URL or from system resources and optionally load its imports.
RuleGrammar	**newRuleGrammar**(java.lang.String name) Create a new `RuleGrammar` for this recognizer with a specified grammar name.
Grammar	**readVendorGrammar**(java.io.InputStream input) Create a new grammar by reading in a grammar stored in a vendor-specific format.
Result	**readVendorResult**(java.io.InputStream output) Read a `Result` object from a stream in a vendor-specific format.
void	**releaseFocus**() Release the speech focus from this `Recognizer`.
void	**removeResultListener**(ResultListener listener) Remove a `ResultListener` from this `Recognizer`.
void	**requestFocus**() Request the speech focus for this `Recognizer` from the underlying speech recognition engine.
void	**suspend**() Temporarily suspend recognition while the application updates grammars prior to a `commitChanges` call.
void	**writeVendorGrammar**(java.io.OutputStream output, Grammar grammar) Store a grammar in a vendor-specific format.
void	**writeVendorResult**(java.io.OutputStream output, Result result) Store a finalized `Result` object in a vendor-specific format so that it can be re-loaded in a future session.

Field Detail

LISTENING

```
public static final long LISTENING
```

`LISTENING` is the bit of state that is set when an `ALLOCATED` `Recognizer` is listening to incoming audio for speech that may match an active grammar but has not yet detected speech.

A `RECOGNIZER_PROCESSING` event is issued to indicate a transition out of the `LISTENING` state and into the `PROCESSING` state.

A `RECOGNIZER_SUSPENDED` event is issued to indicate a transition out of the `LISTENING` state and into the `SUSPENDED` state.

A `CHANGES_COMMITTED` event is issued to indicate a transition into the `LISTENING` state from the `SUSPENDED` state.

See Also:
ALLOCATED, PROCESSING, SUSPENDED, getEngineState, RECOGNIZER_PROCESSING, RECOGNIZER_SUSPENDED, CHANGES_COMMITTED

PROCESSING

```
public static final long PROCESSING
```

`PROCESSING` is the bit of state that is set when an `ALLOCATED` `Recognizer` is producing a `Result` for incoming speech that may match an active grammar.

A `RECOGNIZER_SUSPENDED` event is issued to indicate a transition out of the `PROCESSING` state and into the `SUSPENDED` state when recgonition of a `Result` is completed.

A `RECOGNIZER_PROCESSING` event is issued to indicate a transition into the `PROCESSING` state from the `LISTENING` state when the start of a new result is detected.

See Also:
ALLOCATED, PROCESSING, SUSPENDED, getEngineState, RECOGNIZER_PROCESSING, RECOGNIZER_SUSPENDED

SUSPENDED

```
public static final long SUSPENDED
```

`SUSPENDED` is the bit of state that is set when an `ALLOCATED` `Recognizer` is temporarily suspended while grammar definition and grammar enabled settings are updated. The `Recognizer` enters this state whenever recognition of a `Result` is finalized, and in response to a call to `suspend`.

The primary difference between the `SUSPENDED` and `PAUSED` states of a `Recognizer` is that audio input is buffered in the `SUSPENDED` state. By contrast, the `PAUSED` state indicates that audio input to the `Recognizer` is being ignored. In addition, the `SUSPENDED` state is a temporary state, whereas a `Recognizer` can stay in the `PAUSED` state indefinately.

A `CHANGES_COMMITTED` event is issued to indicate a transition out of the `SUSPENDED` state and into the `LISTENING` state when all changes to grammars are committed to the recognition process.

A `RECOGNIZER_SUSPENDED` event is issued to indicate a transition into the `SUSPENDED` state from either the `LISTENING` state or `PROCESSING` state.

See Also:
ALLOCATED, PROCESSING, SUSPENDED, PAUSED, getEngineState, CHANGES_COMMITTED, RECOGNIZER_SUSPENDED

FOCUS_ON

```
public static final long FOCUS_ON
```

`FOCUS_ON` is the bit of state that is set when an `ALLOCATED` `Recognizer` has the speech focus of the underlying speech recognition engine.

As recognizer focus is gained and lost, a `FOCUS_GAINED` or `FOCUS_LOST` event is issued to indicate the state change. The `requestFocus` and `releaseFocus` methods allow management of speech focus.

See Also:
ALLOCATED, FOCUS_OFF, requestFocus, releaseFocus, FOCUS_GAINED, FOCUS_LOST, getEngineState

---

FOCUS_OFF

```
public static final long FOCUS_OFF
```

`FOCUS_OFF` is the bit of state that is set when an `ALLOCATED` `Recognizer` does not have the speech focus of the underlying speech recognition engine.

As recognizer focus is gained and lost, a `FOCUS_GAINED` or `FOCUS_LOST` event is issued to indicate the state change. The `requestFocus` and `releaseFocus` methods allow management of speech focus.

See Also:
ALLOCATED, FOCUS_ON, requestFocus, releaseFocus, FOCUS_GAINED, FOCUS_LOST, getEngineState

Method Detail

newRuleGrammar

```
public RuleGrammar newRuleGrammar(java.lang.String name)
                           throws java.lang.IllegalArgumentException,
                                  EngineStateError
```

Create a new `RuleGrammar` for this recognizer with a specified grammar name. The new grammar is used for recognition only after changes are committed.

The `newRuleGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`name` - name of the grammar to be created
Throws:
java.lang.IllegalArgumentException - if grammar with name already exists
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

loadJSGF

```
public RuleGrammar loadJSGF(java.io.Reader JSGF)
                                                  throws GrammarException,
                                                         java.io.IOException,
                                                         EngineStateError
```

Create a `RuleGrammar` from Java Speech Grammar Format text provided by the `Reader`. If the grammar contained in the `Reader` already exists, it is over-written. The new grammar is used for recognition only after changes are committed.

It is often useful to load JSGF from a `String` by creating a `StringReader` object with that string.

The caller is responsible for determining the character encoding of the JSGF document. The character encoding information contained in the JSGF header is ignored.

The `loadJSGF` methods operate as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`JSGF` - the Reader from which the grammar text is loaded
Throws:
GrammarException - if the JSGF text contains any errors
java.io.IOException - if an I/O error occurs
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
loadJSGF(URL, String), loadJSGF(URL, String, boolean, boolean, Vector)

---

loadJSGF

```
public RuleGrammar loadJSGF(java.net.URL context,
                            java.lang.String grammarName)
                                       throws GrammarException,
                                              java.net.MalformedURLException,
                                              java.io.IOException,
                                              EngineStateError
```

Load a `RuleGrammar` and its imported grammars from Java Speech Grammar Format text from URLs or from system resources. The loaded grammars are used for recognition only after changes are committed.

The method returns a reference to the named `RuleGrammar`. The method never returns `null` since an exception is thrown if `grammarName` cannot be loaded successfully.

The method attempts to load all imports of `grammarName`, all imports of the imported grammars, and so on. This recursive load stops when grammars are reached that have already been loaded or when no more imports are found. The intent is to ensure that every grammar needed to use the named grammar is loaded.

For example, if we load grammar X, which imports grammar Y, which imports grammars A and B, then all four grammars are loaded. If any of the grammars are already loaded, then it and its imports are not reloaded.

JSGF allows fully-qualified rulename references without a corresponding import statement. This method also attempts to load all grammars referenced by a fully-qualified rulename not already referenced and loaded by a corresponding import statement.

The advanced loadJSGF method provides more control of the loading process. This method is equivalent to:

```
loadJSGF(url, name, true, false, null);
```

(load imports, don't reload existing grammars, don't provide a list of loaded grammars.)

Locating Grammars

The `context` URL parameter is used as the first parameter of the `URL(URL, String)` constructor in the `java.net` package.

The `grammarName` is converted to a grammar filename. The conversion changes each each period character ('.') to a file separator ('/'). The ".gram" suffix is appended to identify the grammar file. The `grammarName` "com.sun.numbers" becomes the filename "com/sun/numbers.gram". This filename is used as the `spec` parameter in the URL constructor.

For example,

```
loadJSGF(new URL("http://www.sun.com/"), "com.sun.numbers")
```

will look for the grammar in the URL "http://www.sun.com/com/sun/numbers.gram" and for its imports in a similar location.

If the derived URL does not exist, `loadJSGF` checks for a system resource with the grammar file using the `ClassLoader.getSystemResource` method. This allows grammars to be included in the `CLASSPATH`.

If the `context` is `null`, the grammars are searched for only as a system resource.

Reading Grammars

For each grammar it loads, the recognizer determines the file's character encoding by reading the JSGF header.

An exception is thrown if (1) any JSGF syntax problems are found, (2) if a grammar found in a URL does not match the expected name, or (3) if a grammar cannot be located as either a URL or system resource.

If an exception is thrown part way through loading a set of grammars the list of loaded grammars not explicitly available. The recognizer does not attempt to remove any partially loaded grammars.

The `loadJSGF` methods operate as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`base` - the URL context from which grammar locations are derived or null to load exclusively from system resources
`grammarName` - the name of the grammar to be loaded

Returns:
a reference to `grammarName`

Throws:
GrammarException - if any loaded JSGF text contains an error
java.net.MalformedURLException - if problem encountered creating a URL
java.io.IOException - if an I/O error occurs
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
loadJSGF(java.io.Reader), loadJSGF(URL, String, boolean, boolean, Vector), URL(java.net.URL, java.net.String)

---

loadJSGF

```
public RuleGrammar loadJSGF(java.net.URL context,
                            java.lang.String grammarName,
                            boolean loadImports,
                            boolean reloadGrammars,
                            java.util.Vector loadedGrammars)
                     throws GrammarException,
                            java.net.MalformedURLException,
                            java.io.IOException,
                            EngineStateError
```

Load a `RuleGrammar` in Java Speech Grammar Format text from a URL or from system resources and optionally load its imports. This method provide an additional control over whether grammars are reloaded even if they have already been loaded, and allows caller to receive a list of all grammars loaded by the `Recognizer`. The loaded grammars are used for recognition only after changes are committed.

The three additional parameters of this method provide the following extensions over the loadJSGF(URL, String) method:

- `loadImports`: if `false`, the method only loads the `RuleGrammar` specified by the `name` parameter.
  If `true`, the method behaves like the loadJSGF(URL, String) method and recursively loads imported grammars.
- `reloadGrammars`: if `true`, the method always loads a `RuleGrammar`, even if it is already loaded into the `Recognizer`. The previous version of the grammar is overwritten.
  If `false`, the method behaves like the loadJSGF(URL, String) method and does not load grammars that are already loaded, or their imports.
- `loadedGrammars`: if non-null, then as the `Recognizer` loads any grammar it appends a reference to that `RuleGrammar` to the `Vector`.

Parameters:
`base` - the URL context from which grammar locations are derived or null to load exclusively from system resources
`grammarName` - the name of the grammar to be loaded
`loadImports` - if `true`, grammars imported by `grammarName` are loaded plus their imports
`reloadGrammars` - if `true` reload all grammars and imports, if `false` do not load grammars already loaded into the `Recognizer`
`loadedGrammars` - if non-null a reference to each loaded `RuleGrammar` is appended as it is loaded

Returns:
a reference to `grammarName`

Throws:
GrammarException - if any loaded JSGF text contains an error
java.net.MalformedURLException - if problem encountered creating a URL
java.io.IOException - if an I/O error occurs
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:
loadJSGF(java.io.Reader), loadJSGF(URL, String)

---

getRuleGrammar

```
public RuleGrammar getRuleGrammar(java.lang.String name)
                           throws EngineStateError
```

Get the `RuleGrammar` with the specified name. Returns `null` if the grammar is not known to the Recognizer.

The `getRuleGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`name` - the name of the grammar to be returned
Returns:
a `RuleGrammar` reference or `null`
Throws:
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

listRuleGrammars

```
public RuleGrammar[] listRuleGrammars()
                                                throws EngineStateError
```

List the `RuleGrammars` known to the `Recognizer`. Returns `null` if there are no grammars.

The `listRuleGrammars` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

deleteRuleGrammar

```
public void deleteRuleGrammar(RuleGrammar grammar)
                       throws java.lang.IllegalArgumentException,
                              EngineStateError
```

Delete a `RuleGrammar` from the `Recognizer`. The grammar deletion only takes effect when all grammar changes are committed.

Recognizers may chose to ignore the deletion of built-in grammars.

The `deleteRuleGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
java.lang.IllegalArgumentException - if the Grammar is not known to the Recognizer
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
commitChanges, CHANGES_COMMITTED

getDictationGrammar

```
public DictationGrammar getDictationGrammar(java.lang.String name)
                                                     throws EngineStateError
```

Return the `DictationGrammar` for a `Recognizer`. Typically, the `name` parameter is `null` to get access to the default `DictationGrammar` for the `Recognizer`.

If the `Recognizer` does not support dictation, or if it does have a `DictationGrammar` with the specified name, then the method returns `null`.

An application can determine whether the `Recognizer` supports dictation by calling the `isDictationGrammarSupported` method of the `RecognizerModeDesc`.

Note: the `name` parameter is provided for future extenion of the API to allow more than one `DictationGrammar` to be defined.

The `getDictationGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
isDictationGrammarSupported

commitChanges

```
public void commitChanges()
                   throws GrammarException,
                          EngineStateError
```

Commit changes to all loaded grammars and all changes of grammar enabling since the last `commitChanges`. Because all changes are applied atomically (all at once) the application does not need to be concerned with intermediate states as it changes grammar definitions and enabling.

The `commitChanges` call first checks that all the loaded grammars are legal. If there are any problems with the current definition of any `RuleGrammar` an exception is thrown. Problems might include undefined rule name, illegal recursion and so on (see the Java Speech Grammar Format Specification and the `GrammarSyntaxDetail` class documentation for details).

The `commitChanges` call is asynchronous (the changes have not necessarily been committed when the call returns). When the changes have been committed, a `CHANGES_COMMITTED` event is issued to all `RecognizerListeners` and to the `GrammarListeners` of all changed `Grammars`.

Immediately following the `CHANGES_COMMITTED` event, a `GRAMMAR_CHANGES_COMMITTED` `GrammarEvent` is issued to the `GrammarListeners` of all changed grammars.

The roll of `commitChanges` in applying grammar changes in described in the documentation for the `Grammar` interface. The effect of the `commitChanges` method upon `Recognizer` states is described above. The use of `suspend` with `commitChanges` and their use for processing asynchronous non-speech events are also described above.

It is not an error to `commitChanges` when no grammars have been changed. However, the `Recognizer` performs state transitions in the same way as when grammars are changed.

The `commitChanges` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
GrammarException - if the loaded grammars contain any logical errors
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
LISTENING, PROCESSING, SUSPENDED, suspend, CHANGES_COMMITTED

---

suspend

```
public void suspend()
            throws EngineStateError
```

Temporarily suspend recognition while the application updates grammars prior to a `commitChanges` call. The `suspend` call places the `Recognizer` in the `SUSPENDED` state. While in that state the incoming audio is buffered. The buffered audio is processed after the recognizer has committed grammar changes and returned to the `LISTENING` state.

The primary difference between the `suspend` and `pause` methods is that audio is buffered while a `Recognizer` is suspended whereas incoming audio is ignored while in the `PAUSED` state. Also, the `SUSPENDED` state should only be visited temporarily, whereas a `Recognizer` can be `PAUSED` indefinately.

The `suspend` method is asynchronous. When the call returns, the recognizer is not necessarily suspended. The `getEngineState` method, or the `RECOGNIZER_SUSPENDED` events can be used to determine when the recognizer is actually suspended.

The use of `suspend` with `commitChanges` for handling atomic grammar changes and for handling asynchronous events are described above.

Calls to `suspend` and `commitChanges` do not nest. A single call to `commitChanges` can release the `Recognizer` after multiple calls to `suspend`.

The `suspend` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states
See Also:
commitChanges, SUSPENDED, RECOGNIZER_SUSPENDED

---

forceFinalize

```
public void forceFinalize(boolean flush)
                  throws EngineStateError
```

If the `Recognizer` is in the `PROCESSING` state (producing a `Result`), force the `Recognizer` to immediately complete processing of that result by finalizing it. It is acceptable behavior for a `Recognizer` to automatically reject the current result.

The `flush` flag indicates whether the recognizer's internally buffered audio should be processed before forcing the finalize. Applications needing immediate cessation of recognition should request a flush. If the force finalize is a response to a user event (e.g. keyboard or mouse press) then the buffer is typically not flushed because incoming speech from a user could be lost.

The state behavior of the `Recognizer` is the same as if the `Result` had been finalized because of end-of-utterance. The `Recognizer` will transition to the `SUSPENDED` state.

The `forceFinalize` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

requestFocus

```
public void requestFocus()
                 throws EngineStateError
```

Request the speech focus for this `Recognizer` from the underlying speech recognition engine. When the focus is received, a `FOCUS_GAINED` event is issued to `RecognizerListeners` and the `Recognizer` changes state from `FOCUS_ON` to `FOCUS_OFF`.

Since one one application may have recognition focus at any time, applications should only request focus when confident that the user is speaking to that application. Speech focus and other focus issues are discussed above in more detail.

It is not an error for an application to request focus for a `Recognizer` that already has speech focus.

The `requestFocus` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:

EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:

releaseFocus, FOCUS_ON, FOCUS_GAINED, getEngineState

---

releaseFocus

```
public void releaseFocus()
                throws EngineStateError
```

Release the speech focus from this `Recognizer`. A `FOCUS_LOST` event is issued to `RecognizerListeners` once the focus is released and the `Recognizer` state changes from `FOCUS_OFF` to `FOCUS_ON`.

Since one one application may have recognition focus at any time, applications should release focus whenever it is not required. Speech focus and other focus issues are discussed above in more detail.

It is not an error for an application to release focus for a `Recognizer` that does not have speech focus.

Focus is implicitly released when a `Recognizer` is deallocated.

The `releaseFocus` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Throws:

EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

See Also:

requestFocus, FOCUS_OFF, FOCUS_LOST, getEngineState

---

addResultListener

```
public void addResultListener(ResultListener listener)
```

Request notifications of all events for all `Result` produced by this `Recognizer`. An application can attach multiple `ResultListeners` to a `Recognizer`. A listener is removed with the `removeResultListener` method.

`ResultListeners` attached to a `Recognizer` are the only `ResultListeners` to receive the `RESULT_CREATED` event and all subsequent events.

`ResultListener` objects can also be attached to any `Grammar` or to any `Result`. A listener attached to the `Grammar` receives all events that match that `Grammar` following a `GRAMMAR_FINALIZED` event. A listener attached to a `Result` receives all events for that result from the time at which the listener is attached.

A `ResultListener` can be attached or removed in any `Engine` state.

See Also:

removeResultListener, addResultListener, addResultListener, RESULT_CREATED, GRAMMAR_FINALIZED

---

removeResultListener

```
public void removeResultListener(ResultListener listener)
```

Remove a `ResultListener` from this `Recognizer`.

A `ResultListener` can be attached or removed in any `Engine` state.

See Also:

addResultListener, removeResultListener, removeResultListener

---

getRecognizerProperties

```
public RecognizerProperties getRecognizerProperties()
```

Return the `RecognizerProperties` object (a JavaBean). The method returns exactly the same object as the `getEngineProperties` method in the `Engine` interface. However, with the `getRecognizerProperties` method, an application does not need to cast the return value.

The `RecognizerProperties` are available in any state of an `Engine`. However, changes only take effect once an engine reaches the `ALLOCATED` state.

Returns:

the `RecognizerProperties` object for this engine

See Also:

getEngineProperties

---

getSpeakerManager

```
public SpeakerManager getSpeakerManager()
                                                          throws java.lang.SecurityException
```

Return an object which provides management of the speakers of a `Recognizer`. Returns `null` if the `Recognizer` does not store speaker data - that is, if it is a *speaker-independent* recognizer in which all speakers are handled the same.

A `getSpeakerManager` returns successfully in any state of a `Recognizer`. The `SpeakerManager` methods that list speakers and set the current speaker operate in any `Recognizer` state but only take effect in the `ALLOCATED` state. This allows an application can set the initial speaker prior to allocating the engine. Other methods of the `SpeakerManager` only operate in the `ALLOCATED` state.

Returns:
the `SpeakerManager` for this `Recognizer`
Throws:
java.lang.SecurityException - if the application does not have `accessSpeakerProfiles` permission

---

readVendorGrammar

```
public Grammar readVendorGrammar(java.io.InputStream input)
                                 throws VendorDataException,
                                        java.io.IOException,
                                        EngineStateError
```

Create a new grammar by reading in a grammar stored in a vendor-specific format. The data could have been stored using the writeVendorGrammar method. The documentation for the writeVendorGrammar method describes the use of vendor grammar formats.

If a grammar of the same name already exists, it is over-written.

The `readVendorGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`input` - `InputStream` from which grammar is loaded
Returns:
reference to the loaded grammar
Throws:
VendorDataException - if the input data format is not known to the `Recognizer`
java.io.IOException - if an I/O error occurs.
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

writeVendorGrammar

```
public void writeVendorGrammar(java.io.OutputStream output,
                               Grammar grammar)
                        throws java.io.IOException,
                               EngineStateError
```

Store a grammar in a vendor-specific format. The data can be re-loaded at a future time by the `readVendorGrammar` method.

The output format will be specific to the type of recognizer that writes it. For example, data written by an Acme Command and Control Recognizer will be readable only by another Acme Command and Control Recognizer or by other recognizers that understand it's format. When portability is required, use the Java Speech Grammar Format.

Why is a Vendor grammar format useful? The recognizer can store information that makes reloading of the grammar faster than when JSGF is used. The recognizer may also store additional information (e.g. pronunciation, statistical and acoustic data) that improve recognition using this grammar. The `writeVendorGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`output` - `OutputStream` where grammar is written
`grammar` - `Grammar` to be written
Throws:
java.io.IOException - if an I/O error occurs.
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

readVendorResult

```
public Result readVendorResult(java.io.InputStream output)
                               throws VendorDataException,
                                      java.io.IOException,
                                      EngineStateError
```

Read a `Result` object from a stream in a vendor-specific format. The return value will include the best-guess tokens, and may include N-best results, audio data, timing data, training information and other data the is optionally provided with a finalized `Result`.

The call returns `null` upon failure. Because result objects are stored in a vendor-specific format they cannot normally be loaded by incompatible recognizers.

The `readVendorResult` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`input` - `InputStream` from which `Result` is loaded
Returns:
reference to the loaded result
Throws:
VendorDataException - if the input data format is not known to the `Recognizer`
java.io.IOException - if an I/O error occurs.
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

writeVendorResult

```
public void writeVendorResult(java.io.OutputStream output,
                              Result result)
                       throws java.io.IOException,
                              ResultStateError,
                              EngineStateError
```

Store a finalized `Result` object in a vendor-specific format so that it can be re-loaded in a future session. All the current information associated with the result is stored. If the application will not need audio data or training information in a future session, they should release that information (through the `FinalResult` interface) before writing the result to reduce storage requirements.

The `writeVendorGrammar` method operates as defined when the `Recognizer` is in the `ALLOCATED` state. The call blocks if the `Recognizer` in the `ALLOCATING_RESOURCES` state and completes when the engine reaches the `ALLOCATED` state. An error is thrown if the `Recognizer` is in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` state.

Parameters:
`output` - `OutputStream` where the result is written
`result` - `Result` to be written

Throws:
ResultStateError - if the `Result` is not in a finalized state
java.io.IOException - if an I/O error occurs.
EngineStateError - if called for a `Recognizer` in the `DEALLOCATED` or `DEALLOCATING_RESOURCES` states

---

Class javax.speech.recognition.RecognizerAdapter

```
java.lang.Object
  |
  +--javax.speech.EngineAdapter
        |
        +--javax.speech.recognition.RecognizerAdapter
```

public class **RecognizerAdapter**
extends EngineAdapter
implements RecognizerListener

The adapter which receives events for a `Recognizer`. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**changesCommitted**(RecognizerEvent e) `CHANGES_COMMITTED` event has been issued as a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state.
void	**focusGained**(RecognizerEvent e) `FOCUS_GAINED` event has been issued as a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state.
void	**focusLost**(RecognizerEvent e) `FOCUS_LOST` event has been issued as a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state.
void	**recognizerProcessing**(RecognizerEvent e) A `RECOGNIZER_PROCESSING` event has been issued as a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state.
void	**recognizerSuspended**(RecognizerEvent e) `RECOGNIZER_SUSPENDED` event has been issued as a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

Methods inherited from class javax.speech.EngineAdapter
engineAllocated, engineAllocatingResources, engineDeallocated, engineDeallocatingResources, engineError, enginePaused, engineResumed

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

recognizerProcessing

`public void **recognizerProcessing**(RecognizerEvent e)`

A `RECOGNIZER_PROCESSING` event has been issued as a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state.

Specified by:
recognizerProcessing in interface RecognizerListener
See Also:
RECOGNIZER_PROCESSING

recognizerSuspended

`public void **recognizerSuspended**(RecognizerEvent e)`

`RECOGNIZER_SUSPENDED` event has been issued as a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

Specified by:
recognizerSuspended in interface RecognizerListener
See Also:
RECOGNIZER_SUSPENDED

changesCommitted

`public void **changesCommitted**(RecognizerEvent e)`

`CHANGES_COMMITTED` event has been issued as a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state.

Following the `CHANGES_COMMITTED` event, a `GRAMMAR_CHANGES_COMMITTED` event is issued to the `GrammarListeners` of each grammar for which a change in its definition or its enabled flag has been committed.

Specified by:
changesCommitted in interface RecognizerListener
See Also:
CHANGES_COMMITTED, GRAMMAR_CHANGES_COMMITTED

focusGained

`public void **focusGained**(RecognizerEvent e)`

`FOCUS_GAINED` event has been issued as a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state.

Specified by:
focusGained in interface RecognizerListener
See Also:
FOCUS_GAINED

focusLost

`public void **focusLost**(RecognizerEvent e)`

`FOCUS_LOST` event has been issued as a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state.

Specified by:
focusLost in interface RecognizerListener
See Also:
FOCUS_LOST

Class javax.speech.recognition.RecognizerAudioAdapter

```
java.lang.Object
  |
  +--javax.speech.AudioAdapter
        |
        +--javax.speech.recognition.RecognizerAudioAdapter
```

---

public class **RecognizerAudioAdapter**
extends AudioAdapter
implements RecognizerAudioListener

Adaptor for a audio events of a recognizer. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**audioLevel**(RecognizerAudioEvent e) `AUDIO_LEVEL` event has occurred indicating a change in the current volume of the audio input to a recognizer.
void	**speechStarted**(RecognizerAudioEvent e) The recognizer has detected a possible start of speech.
void	**speechStopped**(RecognizerAudioEvent e) The recognizer has detected a possible end of speech.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

speechStarted

`public void speechStarted(RecognizerAudioEvent e)`

The recognizer has detected a possible start of speech.

Specified by:
speechStarted in interface RecognizerAudioListener
See Also:
SPEECH_STARTED

---

speechStopped

`public void speechStopped(RecognizerAudioEvent e)`

The recognizer has detected a possible end of speech.

Specified by:
speechStopped in interface RecognizerAudioListener
See Also:
SPEECH_STOPPED

---

audioLevel

`public void audioLevel(RecognizerAudioEvent e)`

`AUDIO_LEVEL` event has occurred indicating a change in the current volume of the audio input to a recognizer.

Specified by:
audioLevel in interface RecognizerAudioListener
See Also:
AUDIO_LEVEL

Class javax.speech.recognition.RecognizerAudioEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.AudioEvent
                    |
                    +--javax.speech.recognition.RecognizerAudioEvent
```

---

public class **RecognizerAudioEvent**
extends AudioEvent

Event issued to indicate detection of speech in the incoming audio stream or to periodically indicate the audio input level. `RecognizerAudioEvents` are issued to each `RecognizerAudioListener` attached to the `AudioManager` for a `Recognizer`.

`RecognizerAudioListener` events are timed with the input audio. In most architectures this indicates that the events occur in near to real time.

`SPEECH_STARTED` and `SPEECH_STOPPED`

The `SPEECH_STARTED` and `SPEECH_STOPPED` events are used to indicate possible detection of speech in the incoming audio. Applications may use this event to display visual feedback to a user indicating that the recognizer is listening - for example, by maintaining a icon indicating microphone status.

It is sometimes difficult to quickly distinguish between speech and other noises (e.g. coughs, microphone bumps), so the `SPEECH_STARTED` event is not always followed by a `Result`.

If a `RESULT_CREATED` is issued for the detected speech, it will usually occur soon after the `SPEECH_STARTED` event but may be delayed for a number of reasons.

- The recognizer may be slower than real time and lag audio input.
- The recognizer may defer issuing a `RESULT_CREATED` until it is confident that it has detected speech that matches one of the active grammars - in some cases the `RESULT_CREATED` may be issued at the end of the spoken sentence.
- The recognizer may be delayed because it is in the `SUSPENDED` state causing it to buffer audio and "catch up" once it returns to the `LISTENING` state.
- Many other reasons.

Some recognizers will allow a user to speak more than one commands without a break. In these cases, a single `SPEECH_STARTED` event may be followed by more than one `RESULT_CREATED` event and result finalization before the `SPEECH_STOPPED` event is issued.

In longer speech (e.g. dictating a paragraph), short pauses in the user's speech may lead to a `SPEECH_STOPPED` event followed by a `SPEECH_STARTED` event as the user resumes speaking. These events do not always indicate that the current result will be finalized.

In short, applications should treat the `SPEECH_STARTED` and `SPEECH_STOPPED` events as operating entirely independently from the result system of a `Recognizer`.

Audio Level Events

An `AUDIO_LEVEL` event indicates a change in the volume of the audio input to a recognizer. The level is defined on a scale from 0 to 1. 0.0 represents silence. 0.25 represents quiet input. 0.75 represents loud input. 1.0 indicates the maximum level.

Audio level events are suitable for display of a visual "VU meter" (like the bar on stereo systems which goes up and down with the volume). Different colors are often used to indicate the different levels: for example, red for loud, green for normal, and blue for background.

Maintaining audio quality is important for reliable recognition. A common problem is a user speaking too loudly or too quietly. The color on a VU meter is one way to provide feedback to the user. Note that a quiet level (below 0.25) does not necessarily indicate that the user is speaking too quietly. The input is also quiet when the user is not speaking.

See Also:
Recognizer, AudioManager, RecognizerAudioListener, RESULT_CREATED, Serialized Form

---

Field Summary

static int	**AUDIO_LEVEL** `AUDIO_LEVEL` event indicates a change in the volume level of the incoming audio.
float	**audioLevel** Audio level defined on a scale from 0 to 1.
static int	**SPEECH_STARTED** The recognizer has detected the possible start of speech in the incoming audio.
static int	**SPEECH_STOPPED** The recognizer has detected the end of speech or noise in the incoming audio that it previously indicated by a `SPEECH_STARTED` event.

Fields inherited from class javax.speech.SpeechEvent	
id	

Fields inherited from class java.util.EventObject	
source	

Constructor Summary

RecognizerAudioEvent(Recognizer source, int id, float audioLevel) Constructs an `RecognizerAudioEvent` with a specified event identifier and audio level.
RecognizerAudioEvent(Recognizer source, int id) Constructs an `RecognizerAudioEvent` with a specified event identifier.

Method Summary

float	**getAudioLevel**() Get the audio input level in the range 0 to 1. A value below 0.25 indicates quiet input with 0.0 being silence.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.AudioEvent
paramString

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

SPEECH_STARTED

`public static final int SPEECH_STARTED`

The recognizer has detected the possible start of speech in the incoming audio. Applications may use this event to display visual feedback to a user indicating that the recognizer is listening.

A detailed description of `SPEECH_STARTED` and `SPEECH_STOPPED` events is provided above.

See Also:
speechStarted, getId, SPEECH_STOPPED

SPEECH_STOPPED

`public static final int SPEECH_STOPPED`

The recognizer has detected the end of speech or noise in the incoming audio that it previously indicated by a `SPEECH_STARTED` event. This event always follows a `SPEECH_STARTED` event.

A detailed description of `SPEECH_STARTED` and `SPEECH_STOPPED` events is provided above.

See Also:
speechStopped, SPEECH_STARTED, getId

AUDIO_LEVEL

`public static final int AUDIO_LEVEL`

`AUDIO_LEVEL` event indicates a change in the volume level of the incoming audio. A detailed description of the `AUDIO_LEVEL` event is provided above.

See Also:
audioLevel

---

audioLevel

`protected float audioLevel`

Audio level defined on a scale from 0 to 1.

See Also:
getAudioLevel

Constructor Detail

RecognizerAudioEvent

```
public RecognizerAudioEvent(Recognizer source,
                            int id,
                            float audioLevel)
```

Constructs an `RecognizerAudioEvent` with a specified event identifier and audio level. The `audioLevel` should be `0.0` for `SPEECH_STARTED` and `SPEECH_STOPPED` events.

Parameters:

`source` - the `Recognizer` that issued the event
`id` - the identifier for the event type
`audioLevel` - the audio level for this event

RecognizerAudioEvent

```
public RecognizerAudioEvent(Recognizer source,
                            int id)
```

Constructs an `RecognizerAudioEvent` with a specified event identifier. The `audioLevel` is set to `0.0`.

Parameters:

`source` - the `Recognizer` that issued the event
`id` - the identifier for the event type

Method Detail

getAudioLevel

```
public float getAudioLevel()
```

Get the audio input level in the range 0 to 1. A value below 0.25 indicates quiet input with 0.0 being silence. A value above 0.75 indicates loud input with 1.0 indicating the maximum level.

The level is provided only for the `AUDIO_LEVEL` event type. The level should be ignored for `SPEECH_STARTED` and `SPEECH_STOPPED` events.

See Also:

AUDIO_LEVEL

paramString

```
public java.lang.String paramString()
```

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:

a string identifying the event

Overrides:

paramString in class AudioEvent

Interface javax.speech.recognition.RecognizerAudioListener

Implementing Classes:
RecognizerAudioAdapter

---

public abstract interface **RecognizerAudioListener**
extends AudioListener

Extends the set of audio event of an engine for a recognizer by adding a audio level event.

The `RecognizerAudioAdapter` class provided a trivial implementation of this listener.

See Also:
RecognizerAudioAdapter

Method Summary

void	**audioLevel**(RecognizerAudioEvent e) `AUDIO_LEVEL` event has occurred indicating a change in the current volume of the audio input to a recognizer.
void	**speechStarted**(RecognizerAudioEvent e) The recognizer has detected a possible start of speech.
void	**speechStopped**(RecognizerAudioEvent e) The recognizer has detected a possible end of speech.

Method Detail

speechStarted

`public void speechStarted(RecognizerAudioEvent e)`

The recognizer has detected a possible start of speech.

See Also:
SPEECH_STARTED

speechStopped

`public void speechStopped(RecognizerAudioEvent e)`

The recognizer has detected a possible end of speech.

See Also:
SPEECH_STOPPED

audioLevel

`public void audioLevel(RecognizerAudioEvent e)`

`AUDIO_LEVEL` event has occurred indicating a change in the current volume of the audio input to a recognizer.

See Also:
AUDIO_LEVEL

Class javax.speech.recognition.RecognizerEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.EngineEvent
                    |
                    +--javax.speech.recognition.RecognizerEvent
```

---

public class **RecognizerEvent**
extends EngineEvent

Event issued by `Recognizer` through `RecognizerListener`. Inherits the following event types from `EngineEvent`: `ENGINE_ALLOCATED`, `ENGINE_DEALLOCATED`, `ENGINE_ALLOCATING_RESOURCES`, `ENGINE_DEALLOCATING_RESOURCES`, `ENGINE_PAUSED`, `ENGINE_RESUMED`.

The source object for any `RecognizerEvent` is the `Recognizer`.

See Also:
EngineEvent, Recognizer, RecognizerListener, Serialized Form

Field Summary

static int	**CHANGES_COMMITTED** `CHANGES_COMMITTED` event is issued when a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state.
static int	**FOCUS_GAINED** `FOCUS_GAINED` event is issued when a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state.
static int	**FOCUS_LOST** `FOCUS_LOST` event is issued when a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state.
GrammarException	**grammarException** Non-null if any error is detected in a grammar's definition while producing a `CHANGES_COMMITTED` event.
static int	**RECOGNIZER_PROCESSING** `RECOGNIZER_PROCESSING` event is issued when a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state to indicate that it is actively processing a recognition `Result`.
static int	**RECOGNIZER_SUSPENDED** `RECOGNIZER_SUSPENDED` event is issued when a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

Fields inherited from class javax.speech.EngineEvent
ENGINE_ALLOCATED, ENGINE_ALLOCATING_RESOURCES, ENGINE_DEALLOCATED, ENGINE_DEALLOCATING_RESOURCES, ENGINE_PAUSED, ENGINE_RESUMED, newEngineState, oldEngineState

Fields inherited from class javax.speech.SpeechEvent
id

Fields inherited from class java.util.EventObject
source

Constructor Summary

RecognizerEvent(Recognizer source, int id, long oldEngineState, long newEngineState, GrammarException grammarException) Construct a `RecognizerEvent` with a specified event source, event identifier, old and new states, and optionally a `GrammarException` for a `CHANGES_COMMITTED` event.

Method Summary

GrammarException	**getGrammarException**() Returns non-null for a `CHANGES_COMMITTED` event if an error is found in the grammar definition.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.EngineEvent
getNewEngineState, getOldEngineState, paramString

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

RECOGNIZER_PROCESSING

`public static final int` **`RECOGNIZER_PROCESSING`**

`RECOGNIZER_PROCESSING` event is issued when a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state to indicate that it is actively processing a recognition `Result`. The transition is triggered when the recognizer detects speech in the incoming audio stream that may match and active grammar. The transition occurs immediately before the `RESULT_CREATED` is issued to `ResultListeners`.

See Also:
recognizerProcessing, LISTENING, PROCESSING, RESULT_CREATED, getEngineState, getId

RECOGNIZER_SUSPENDED

`public static final int` **`RECOGNIZER_SUSPENDED`**

`RECOGNIZER_SUSPENDED` event is issued when a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

A transition from the `LISTENING` state to the `SUSPENDED` state is triggered by a call to either the `suspend` method or the `commitChanges` method.

A transition from the `PROCESSING` state to the `SUSPENDED` state is triggered by the finalization of the result currently being recognized. In this instance, the `RECOGNIZER_SUSPENDED` event is followed immediately by either the `RESULT_ACCEPTED` or `RESULT_REJECTED` event that finalizes the result.

See Also:
recognizerSuspended, LISTENING, PROCESSING, SUSPENDED, suspend, commitChanges, getEngineState, getId, RESULT_ACCEPTED, RESULT_REJECTED

CHANGES_COMMITTED

`public static final int` **`CHANGES_COMMITTED`**

`CHANGES_COMMITTED` event is issued when a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state. This state transition takes place when changes to the definition and enabled state of all a recognizer’s grammars have been applied. The new grammar definitions are used as incoming speech is recognized in the `LISTENING` and `PROCESSING` states of the `Recognizer`.

Immediately following the `CHANGES_COMMITTED` event, `GRAMMAR_CHANGES_COMMITTED` events are issued to the `GrammarListeners` of each changed `Grammar`.

If any errors are detected in any grammar’s definition during the commit, a `GrammarException` is provided with this event. The `GrammarException` is also included with the `GRAMMAR_CHANGES_COMMITTED` event to `Grammar` with the error. The `GrammarException` has the same function as the `GrammarException` thrown on the `commitChanges` method.

The causes and timing of the `CHANGES_COMMITTED` event are described with the state transition documentation for a `Recognizer` with the the committing changes documentation for a `Grammar`.

See Also:
changesCommitted, commitChanges, LISTENING, SUSPENDED, GRAMMAR_CHANGES_COMMITTED, getEngineState, getId

FOCUS_GAINED

`public static final int` **`FOCUS_GAINED`**

`FOCUS_GAINED` event is issued when a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state. This event typically occurs as a result of a call the the `requestFocus` method of the `Recognizer`.

The event indicates that the `FOCUS_ON` bit of the engine state is set.

Since recognizer focus is a key factor in the activation policy for grammars, a `FOCUS_GAINED` event is followed by a `GRAMMAR_ACTIVATED` event to the `GrammarListeners` of each `Grammar` that is activated. Activation conditions and the role of recognizer focus are detailed in the documentation for the `Grammar` interface.

See Also:
focusGained, FOCUS_LOST, FOCUS_ON, requestFocus, GRAMMAR_ACTIVATED, Grammar

FOCUS_LOST

`public static final int FOCUS_LOST`

`FOCUS_LOST` event is issued when a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state. This event may occur as a result of a call to the `releaseFocus` method of the `Recognizer` or because another application has requested recognizer focus.

The event indicates that the `FOCUS_OFF` bit of the engine state is set.

Since recognizer focus is a key factor in the activation policy for grammars, a `FOCUS_LOST` event is followed by a `GRAMMAR_DEACTIVATED` event to the `GrammarListeners` of each `Grammar` that loses activatation. Activation conditions and the role of recognizer focus are detailed in the documentation for the `Grammar` interface.

See Also:
focusLost, FOCUS_GAINED, FOCUS_OFF, releaseFocus, GRAMMAR_DEACTIVATED, Grammar

---

grammarException

`protected GrammarException grammarException`

Non-null if any error is detected in a grammar's definition while producing a `CHANGES_COMMITTED` event. `null` for other event types. The exception serves the same functional role as the `GrammarException` thrown on the `commitChanges` method.

See Also:
getGrammarException, commitChanges

Constructor Detail

RecognizerEvent

```
public RecognizerEvent(Recognizer source,
                       int id,
                       long oldEngineState,
                       long newEngineState,
                       GrammarException grammarException)
```

Construct a `RecognizerEvent` with a specified event source, event identifier, old and new states, and optionally a `GrammarException` for a `CHANGES_COMMITTED` event.

Parameters:
`source` - the `Recognizer` that issued the event
`id` - the identifier for the event type
`oldEngineState` - engine state prior to this event
`newEngineState` - engine state following this event
`grammarException` - non-null if an error is detected during `CHANGES_COMMITTED`

Method Detail

getGrammarException

`public GrammarException getGrammarException()`

Returns non-null for a `CHANGES_COMMITTED` event if an error is found in the grammar definition. The exception serves the same functional role as the `GrammarException` thrown on the `commitChanges` method.

See Also:
commitChanges

---

paramString

`public java.lang.String paramString()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class EngineEvent

Interface javax.speech.recognition.RecognizerListener

Implementing Classes:
RecognizerAdapter

---

public abstract interface **RecognizerListener**
extends EngineListener

Defines an extension to the `EngineListener` interface for specific events associated with a `Recognizer`. A `RecognizerListener` object is attached to and removed from a `Recognizer` by calls to the `addEngineListener` and `removeEngineListener` methods (which `Recognizer` inherits from the `Engine` interface).

The `RecognizerAdapter` class provides a trivial implementation of this interface.

See Also:
RecognizerAdapter, RecognizerEvent, Recognizer, Engine, addEngineListener, removeEngineListener

---

Method Summary

void	**changesCommitted**(RecognizerEvent e) A `CHANGES_COMMITTED` event has been issued as a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state and resumed recognition.
void	**focusGained**(RecognizerEvent e) `FOCUS_GAINED` event has been issued as a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state.
void	**focusLost**(RecognizerEvent e) `FOCUS_LOST` event has been issued as a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state.
void	**recognizerProcessing**(RecognizerEvent e) A `RECOGNIZER_PROCESSING` event has been issued as a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state.
void	**recognizerSuspended**(RecognizerEvent e) A `RECOGNIZER_SUSPENDED` event has been issued as a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

Method Detail

recognizerProcessing

public void **recognizerProcessing**(RecognizerEvent e)

A `RECOGNIZER_PROCESSING` event has been issued as a `Recognizer` changes from the `LISTENING` state to the `PROCESSING` state.

See Also:
RECOGNIZER_PROCESSING

---

recognizerSuspended

public void **recognizerSuspended**(RecognizerEvent e)

A `RECOGNIZER_SUSPENDED` event has been issued as a `Recognizer` changes from either the `LISTENING` state or the `PROCESSING` state to the `SUSPENDED` state.

A `Result` finalization event (either a `RESULT_ACCEPTED` or `RESULT_REJECTED` event) is issued immediately following the `RECOGNIZER_SUSPENDED` event.

See Also:
RECOGNIZER_SUSPENDED

---

changesCommitted

public void **changesCommitted**(RecognizerEvent e)

A `CHANGES_COMMITTED` event has been issued as a `Recognizer` changes from the `SUSPENDED` state to the `LISTENING` state and resumed recognition.

The `GRAMMAR_CHANGES_COMMITTED` event is issued to the `GrammarListeners` of all changed grammars immediately following the `CHANGES_COMMITTED` event.

See Also:
CHANGES_COMMITTED, GRAMMAR_CHANGES_COMMITTED

---

focusGained

public void **focusGained**(RecognizerEvent e)

`FOCUS_GAINED` event has been issued as a `Recognizer` changes from the `FOCUS_OFF` state to the `FOCUS_ON` state. A `FOCUS_GAINED` event typically follows a call to `requestFocus` on a `Recognizer`.

The `GRAMMAR_ACTIVATED` event is issued to the `GrammarListeners` of all activated grammars immediately following this `RecognizerEvent`.

See Also:
FOCUS_GAINED, GRAMMAR_ACTIVATED, requestFocus

focusLost

`public void` **`focusLost`**`(RecognizerEvent e)`

`FOCUS_LOST` event has been issued as a `Recognizer` changes from the `FOCUS_ON` state to the `FOCUS_OFF` state. A `FOCUS_LOST` event may follow a call to `releaseFocus` on a `Recognizer` or follow a request for focus by another application.

The `GRAMMAR_DEACTIVATED` event is issued to the `GrammarListeners` of all deactivated grammars immediately following this `RecognizerEvent`.

See Also:
FOCUS_LOST, GRAMMAR_DEACTIVATED, releaseFocus

Class javax.speech.recognition.RecognizerModeDesc

```
java.lang.Object
  |
  +--javax.speech.EngineModeDesc
        |
        +--javax.speech.recognition.RecognizerModeDesc
```

public class **RecognizerModeDesc**
extends EngineModeDesc

`RecognizerModeDesc` extends the `EngineModeDesc` with properties that are specific to speech recognizers. A `RecognizerModeDesc` inherits engine name, mode name and `Locale` from `EngineModeDesc`. `RecognizerModeDesc` adds three features:

- A `Boolean` indicating whether dictation is supported
- List of the recognizer’s speaker profiles
- Speaker profile that will be loaded when a recognizer is started (not used in selection)

Like `EngineModeDesc`, there are two types of `RecognizerModeDesc`: those created by an application for use in engine selection, and those created by an engine which descibe a particular mode of operation of the engine. Descriptors provided by engines are obtained through the `availableRecognizers` method of the `Central` class. These descriptors must have all their features defined. A descriptor created by an application may make any or all of the features `null` to indictae a "don’t care" value (null features are ignored in engine selection).

[Note: the `Boolean` *"is running"* feature is a full object rather than a `boolean` primitive so that it can represent true, false and "don’t care".]

Applications can modify application-created descriptors in any way. Applications should never modify the features of a `RecognizerModeDesc` provided by an engine (i.e. returned by the `availableRecognizers` method).

Engine creation is described in the documentation for the `Central` class.

See Also:
Central, SpeakerManager, SpeakerProfile

Constructor Summary

RecognizerModeDesc() Construct a descriptor with `null` values ("don’t care") for all recognizer features.
RecognizerModeDesc(java.util.Locale locale, java.lang.Boolean dictationGrammarSupported) Create a `RecognizerModeDesc` given a `Locale` and the dictation flag.
RecognizerModeDesc(java.lang.String engineName, java.lang.String modeName, java.util.Locale locale, java.lang.Boolean running, java.lang.Boolean dictationGrammarSupported, SpeakerProfile[] profiles) Create a fully-specified descriptor.

Method Summary

void	**addSpeakerProfile**(SpeakerProfile profile) Add a speaker profile to the `SpeakerProfile` array.
boolean	**equals**(java.lang.Object anObject) True if and only if the input parameter is not `null` and is a `RecognizerModeDesc` with equal values of dictationGrammarSupported and all speaker profiles.
SpeakerProfile[]	**getSpeakerProfilesImpl**() Version of `getSpeakerProfiles` that performs the operation.
SpeakerProfile[]	**getSpeakerProfiles**() Returns the list of speaker profiles known to this mode of this recognition engine.
java.lang.Boolean	**isDictationGrammarSupported**() Test whether this engine mode provides a `DictationGrammar`.
boolean	**match**(EngineModeDesc require) Determine whether a `RecognizerModeDesc` has all the features defined in the `require` object.
void	**setDictationGrammarSupported**(java.lang.Boolean dictationGrammarSupported) Set the dictationGrammarSupported parameter.
void	**setSpeakerProfiles**(SpeakerProfile[] speakers) Set the list of speaker profiles.

Methods inherited from class javax.speech.EngineModeDesc
equals, getEngineName, getLocale, getModeName, getRunning, match, setEngineName, setLocale, setModeName, setRunning

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

RecognizerModeDesc

```
public RecognizerModeDesc()
```

Construct a descriptor with `null` values ("don't care") for all recognizer features.

RecognizerModeDesc

```
public RecognizerModeDesc(java.util.Locale locale,
                          java.lang.Boolean dictationGrammarSupported)
```

Create a `RecognizerModeDesc` given a `Locale` and the dictation flag. The speaker profiles array and other features are all `null`.

RecognizerModeDesc

```
public RecognizerModeDesc(java.lang.String engineName,
                          java.lang.String modeName,
                          java.util.Locale locale,
                          java.lang.Boolean running,
                          java.lang.Boolean dictationGrammarSupported,
                          SpeakerProfile[] profiles)
```

Create a fully-specified descriptor.

Method Detail

isDictationGrammarSupported

```
public java.lang.Boolean isDictationGrammarSupported()
```

Test whether this engine mode provides a `DictationGrammar`. The value may be `TRUE`, `FALSE` or `null`. A `null` value means "don't care".

---

setDictationGrammarSupported

```
public void setDictationGrammarSupported(java.lang.Boolean dictationGrammarSupported)
```

Set the dictationGrammarSupported parameter. The value may be `TRUE`, `FALSE` or `null`. A `null` value means "don't care".

---

getSpeakerProfiles

```
public final SpeakerProfile[] getSpeakerProfiles()
                                          throws java.lang.SecurityException
```

Returns the list of speaker profiles known to this mode of this recognition engine. Returns `null` if speaker training is not supported (`SpeakerManager` not implemented). Returns zero-length array if speaker training is supported but no speaker profiles have been constructed yet.

The list of speaker profiles is the same as returned by the the `listKnownSpeakers` method of `SpeakerManager` if this engine is running.

Throws:
java.lang.SecurityException - if the application does not have `accessSpeakerProfiles` permission
See Also:
listKnownSpeakers

getSpeakerProfilesImpl

`protected SpeakerProfile[] getSpeakerProfilesImpl()`

Version of `getSpeakerProfiles` that performs the operation. This method can be overridden in sub-classes. However, application can only call the `getSpeakerProfiles` method which does a security check.

See Also:
getSpeakerProfiles

setSpeakerProfiles

`public void setSpeakerProfiles(SpeakerProfile[] speakers)`

Set the list of speaker profiles. May be `null`.

addSpeakerProfile

`public void addSpeakerProfile(SpeakerProfile profile)`

Add a speaker profile to the `SpeakerProfile` array.

match

`public boolean match(EngineModeDesc require)`

Determine whether a `RecognizerModeDesc` has all the features defined in the `require` object. Strings in `require` which are either `null` or zero-length ("") are not tested, including those in the `Locale`. All string comparisons are exact (case-sensitive).

The parameters are used as follows:

- If the require parameter is an EngineModeDesc then only the `EngineModeDesc` features are tested (engine name, mode name, locale).
- If the require parameter is a `RecognizerModeDesc` (or sub-class) then the grammar supported flags and required speakers are tested as follows.
    - Every speaker profile in the required set must be matched by a profile in the tested object. A `null` require speakers value is ignored.
    - Match dictation supported `Boolean` value if the required value is `null` or if exact boolean match.

Overrides:
match in class EngineModeDesc

equals

`public boolean equals(java.lang.Object anObject)`

True if and only if the input parameter is not `null` and is a `RecognizerModeDesc` with equal values of dictationGrammarSupported and all speaker profiles.
Overrides:
equals in class EngineModeDesc

Interface javax.speech.recognition.RecognizerProperties

public abstract interface **RecognizerProperties**
extends EngineProperties

Enables control of the properties of a `Recognizer`. The `RecognizerProperties` object is obtained by calling the `getEngineProperties` method of the `Recognizer` (inherited from the `Engine` interface).

Because `RecognizerProperties` extends the `EngineProperties` interface, it inherits the following behavior (described in detail in the `EngineProperties` documentation):

- Each property has a get and set method. (JavaBeans property method patterns)
- Engines may ignore changes to properties or apply maximum and minimum limits. If an engine does not apply a property change request, the set method throws a `PropertyVetoException`.
- Calls to set methods may be asynchronous (they may return before the property change takes effect). The `Engine` will apply a change as soon as possible.
- The get methods return the current setting - not a pending value.
- A `PropertyChangeListener` may be attached to receive property change events.
- Where appropriate, property settings are persistent across sessions.

Per-Speaker Properties

For recognizers that maintain speaker data (recognizers that implement the `SpeakerManager` interface) the `RecognizerProperties` should be stored and loaded as part of the speaker data. Thus, when the current speaker is changed through the `SpeakerManager` interface, the properties of the new speaker should be loaded.

See Also:
EngineProperties, SpeakerManager

Method Summary

float	**getCompleteTimeout**() Get the "CompleteTimeout" property.
float	**getConfidenceLevel**() Get the recognizer’s "ConfidenceLevel" property.
float	**getIncompleteTimeout**() Get the "IncompleteTimeout" property.
int	**getNumResultAlternatives**() Get the "NumResultAlternatives" property.
float	**getSensitivity**() Get the "Sensitivity" property.
float	**getSpeedVsAccuracy**() Get the "SpeedVsAccuracy" property.
boolean	**isResultAudioProvided**() Get the `"ResultAudioProvided"` property.
boolean	**isTrainingProvided**() Get the `TrainingProvided` property.
void	**setCompleteTimeout**(float timeout) Set the `CompleteTimeout` property in seconds.
void	**setConfidenceLevel**(float confidenceLevel) Set the recognizer’s "ConfidenceLevel" property.
void	**setIncompleteTimeout**(float timeout) Set the `IncompleteTimeout` property in seconds.
void	**setNumResultAlternatives**(int num) Set the `"NumResultAlternatives"` property.
void	**setResultAudioProvided**(boolean audio) Set the `"ResultAudioProvided"` property.
void	**setSensitivity**(float sensitivity) Set the "Sensitivity" property.
void	**setSpeedVsAccuracy**(float speedVsAccuracy) Get the "SpeedVsAccuracy" property.
void	**setTrainingProvided**(boolean trainingProvided) Set the `TrainingProvided` property.

Method Detail

getConfidenceLevel

```
public float getConfidenceLevel()
```

Get the recognizer's "ConfidenceLevel" property.

See Also:
setConfidenceLevel

---

setConfidenceLevel

```
public void setConfidenceLevel(float confidenceLevel)
                       throws java.beans.PropertyVetoException
```

Set the recognizer's "ConfidenceLevel" property. The confidence level value can very between 0.0 and 1.0. A value of 0.5 is the default for the recognizer. A value of 1.0 requires the recognizer to have maximum confidence in its results or otherwise reject them. A value of 0.0 requires only low confidence so fewer results are rejected.

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value
See Also:
getConfidenceLevel

---

getSensitivity

```
public float getSensitivity()
```

Get the "Sensitivity" property.

See Also:
setSensitivity

---

setSensitivity

```
public void setSensitivity(float sensitivity)
                   throws java.beans.PropertyVetoException
```

Set the "Sensitivity" property. The sensitivity level can vary between 0.0 and 1.0. A value of 0.5 is the default for the recognizer. A value of 1.0 gives maximum sensitivity and makes the recognizer sensitive to quiet input but more sensitive to noise. A value of 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 extreme cases where the automatic settings are not adequate.

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value

See Also:
getSensitivity

---

getSpeedVsAccuracy

```
public float getSpeedVsAccuracy()
```

Get the "SpeedVsAccuracy" property.

See Also:
setSpeedVsAccuracy

---

setSpeedVsAccuracy

```
public void setSpeedVsAccuracy(float speedVsAccuracy)
                       throws java.beans.PropertyVetoException
```

Get the "SpeedVsAccuracy" property. The default value is 0.5 and is the factory-defined compromise of speed and accuracy. A value of 0.0 minimizes response time. A value of 1.0 maximizes recognition accuracy.

Why are speed and accuracy a trade-off? A recognizer determines what a user says by testing different possible sequence of words (with legal sequences being defined by the active grammars). If the recognizer tests more sequences it is more likely to find the correct sequence, but there is additional processing so it is slower. Increasing grammar complexity and decreasing the computer power both make this trade-off more important. Conversely, a simpler grammar or more powerful computer make the trade-off less important.

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value
See Also:
getSpeedVsAccuracy

---

getCompleteTimeout

```
public float getCompleteTimeout()
```

Get the "CompleteTimeout" property.
See Also:
setCompleteTimeout

---

setCompleteTimeout

```
public void setCompleteTimeout(float timeout)
                       throws java.beans.PropertyVetoException
```

Set the `CompleteTimeout` property in seconds. This timeout value determines the length of silence required following user speech before the recognizer finalizes a result (with an `RESULT_ACCEPTED` or `RESULT_REJECTED` event). The complete timeout is used when the speech is a complete match an active `RuleGrammar`. By contrast, the incomplete timeout is used when the speech is an incomplete match to an active grammar.

A long complete timeout value delays the result completion and therefore makes the computer's response slow. A short complete timeout may lead to an utterance being broken up inappropriately. Reasonable complete timeout values are typically in the range of 0.3 seconds to 1.0 seconds.

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value
See Also:
getCompleteTimeout

---

getIncompleteTimeout

```
public float getIncompleteTimeout()
```

Get the "IncompleteTimeout" property.

See Also:
setIncompleteTimeout

---

setIncompleteTimeout

```
public void setIncompleteTimeout(float timeout)
                         throws java.beans.PropertyVetoException
```

Set the `IncompleteTimeout` property in seconds. The timeout value determines the required length of silence following user speech after which a recognizer finalizes a result.

The incomplete timeout applies when the speech prior to the silence is an incomplete match of the active `RuleGrammars`. In this case, once the timeout is triggered, the partial result is rejected (with a `RESULT_REJECTED` event).

The incomplete timeout also applies when the speech prior to the silence is a complete match of an active grammar, but where it is possible to speak further and still match the grammar. For example, in a grammar for digit sequences for telephone numbers it might be legal to speak either 7 or 10 digits. If the user pauses briefly after speaking 7 digits then the incomplete timeout applies because the user may then continue with a further 3 digits.

By contrast, the complete timeout is used when the speech is a complete match to an active `RuleGrammar` and no further words can be spoken.

A long complete timeout value delays the result completion and therefore makes the computer's response slow. A short incomplete timeout may lead to an utterance being broken up inappropriately.

The incomplete timeout is usually longer than the complete timeout to allow users to pause mid-utterance (for example, to breathe).

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value
See Also:
getIncompleteTimeout

---

getNumResultAlternatives

```
public int getNumResultAlternatives()
```

Get the "NumResultAlternatives" property.
See Also:
setNumResultAlternatives

---

setNumResultAlternatives

```
public void setNumResultAlternatives(int num)
                         throws java.beans.PropertyVetoException
```

Set the `"NumResultAlternatives"` property. This property indicates the preferred maximum number of N-best alternatives in `FinalDictationResult` and `FinalRuleResult` objects. A value of 0 or 1 indicates that the application does not want alternatives; that is, it only wants the best guess.

Recognizers are not required to provide this number of alternatives for every result and the number of alternatives may vary from result to result. Recognizers should only provide alternative tokens which are considered reasonable guesses: that is, the alternatives should be above the `ConfidenceLevel` set through this interface (unless the `Result` is rejected).

Providing alternatives requires additional computing power. Applications should only request the number of alternatives that they are likely to use.

Throws:
java.beans.PropertyVetoException - if the recognizer rejects or limits the new value
See Also:
getNumResultAlternatives, getAlternativeTokens(ResultToken, ResultToken, int), getAlternativeTokens(int)

---

isResultAudioProvided

```
public boolean isResultAudioProvided()
```

Get the `"ResultAudioProvided"` property.

See Also:
setResultAudioProvided

setResultAudioProvided

```
public void setResultAudioProvided(boolean audio)
                            throws java.beans.PropertyVetoException
```

Set the `"ResultAudioProvided"` property. If set to `true`, the recognizer is requested to provide audio with `FinalResult` objects. If available, the audio is provided through the `getAudio` methods of the `FinalResult` interface.

Some recognizers that can provide audio for a `FinalResult` cannot provide audio for all results. Applications need test audio available individually for results. (For example, a recognizer might only provide audio for dictation results.)

A `Recognizer` that does not provide audio for any results throws a `PropertyVetoException` when an app attempts to set the value to `true`.

Throws:

java.beans.PropertyVetoException - if the recognizer rejects or limits the new value

See Also:

isResultAudioProvided, getAudio(), getAudio(ResultToken, ResultToken)

---

isTrainingProvided

```
public boolean isTrainingProvided()
```

Get the `TrainingProvided` property.

See Also:

setTrainingProvided

---

setTrainingProvided

```
public void setTrainingProvided(boolean trainingProvided)
                         throws java.beans.PropertyVetoException
```

Set the `TrainingProvided` property. If `true`, request a recognizer to provide training information for `FinalResult` objects through the `tokenCorrection` method.

Not all recognizers support training. Also, recognizers that do support training are not required to support training data for all results. For example, a recognizer might only produce training data with dictation results. A `Recognizer` that does not support training throws a `PropertyVetoException` when an app attempts to set the value to `true`.

Throws:

java.beans.PropertyVetoException - if the recognizer rejects the new value

See Also:

isTrainingProvided, tokenCorrection

Interface javax.speech.recognition.Result

Subinterfaces:

FinalResult

public abstract interface **Result**

A `Result` is issued by a `Recognizer` as it recognizes an incoming utterance that matches an active `Grammar`. The `Result` interface provides the application with access to the following information about a recognized utterance:

1. A sequence of finalized tokens (words) that have been recognized,
2. A sequence of unfinalized tokens,
3. Reference to the grammar matched by the result,
4. The result state: `UNFINALIZED`, `ACCEPTED` or `REJECTED`.

Multiple Result Interfaces

Every `Result` object provided by a `Recognizer` implements both the `FinalRuleResult` and `FinalDictationResult` interfaces. Thus, by extension every result also implements the `FinalResult` and `Result` interfaces.

These multiple interfaces are designed to explicitly indicate (a) what information is available at what times in the result life-cycle and (b) what information is available for different types of results. Appropriate casting of results allows compile-time checking of result-handling code and fewer bugs.

The `FinalResult` extends the `Result` interface. It provides access to the additional information about a result that is available once it has been finalized (once it is in either of the `ACCEPTED` or `REJECTED` states). Calling any method of the `FinalResult` interface for a result in the `UNFINALIZED` state causes a `ResultStateError` to be thrown.

The `FinalRuleResult` extends the `FinalResult` interface. It provides access to the additional information about a finalized result that matches a `RuleGrammar`. Calling any method of the `FinalRuleResult` interface for a non-finalized result or a result that matches a `DictationGrammar` causes a `ResultStateError` to be thrown.

The `FinalDictationResult` also extends the `FinalResult` interface. It provides access to the additional information about a finalized result that matches a `DictationGrammar`. Calling any method of the `FinalDictationResult` interface for a non-finalized result or a result that matches a `RuleGrammar` causes a `ResultStateError` to be thrown.

Note: every result implements both the `FinalRuleResult` and `FinalDictationResult` interfaces even though the result will match either a `RuleGrammar` or `DictationGrammar`, but *never both*. The reason for this is that when the result is created (`RESULT_CREATED` event), the grammar is not always known.

Result States

The separate interfaces determine what information is available for a result in the different stages of its life-cycle. The state of a `Result` is determined by calling the `getResultState` method. The three possible states are `UNFINALIZED`, `ACCEPTED` and `REJECTED`.

A new result starts in the `UNFINALIZED` state. When the result is *finalized* is moves to either the `ACCEPTED` or `REJECTED` state. An accepted or rejected result is termed a *finalized* result. All values and information regarding a finalized result are fixed (excepting that audio and training information may be released).

Following are descriptions of a result object in each of the three states including information on which interfaces can be used in each state.

`getResultState() == Result.UNFINALIZED`

- Recognition of the result is in progress.
- A new result is created with a `RESULT_CREATED` event that is issued to each `ResultListener` attached to a `Recognizer`. The new result is created in in the `UNFINALIZED` state.
- A result remains in the `UNFINALIZED` state until it is *finalized* by either a `RESULT_ACCEPTED` or `RESULT_REJECTED` event.
- Applications should only call the methods of the `Result` interface. A `ResultStateError` is issued on calls to the methods of `FinalResult`, `FinalRuleResult` and `FinalDictationResult` interfaces.
- Events 1: zero or more `RESULT_UPDATED` events may be issued as (a) tokens are finalized, or (b) as the unfinalized tokens changes.
- Events 2: one `GRAMMAR_FINALIZED` event must be issued in the `UNFINALIZED` state before result finalization by an `RESULT_ACCEPTED` event. (Not required if a result is rejected.)
- Events 3: the `GRAMMAR_FINALIZED` event is optional if the result is finalized by a `RESULT_REJECTED` event. (It is not always possible for a recognizer to identify a best-match grammar for a rejected result.)
- Prior to the `GRAMMAR_FINALIZED` event, the `getGrammar` returns `null`. Following the `GRAMMAR_FINALIZED` event the `getGrammar` method returns a non-null reference to the active `Grammar` that is matched by this result.
- `numTokens` returns the number of finalized tokens. While in the `UNFINALIZED` this number may increase as `ResultEvent.RESULT_UPDATED` events are issued.
- The best guess for each finalized token is available through `getBestToken(int num)`. The best guesses are guaranteed not to change through the remaining life of the result.
- `getUnfinalizedTokens` may return zero or more tokens and these may change at any time when a `ResultEvent.RESULT_UPDATED` event is issued.

`getResultState() == Result.ACCEPTED`

- Recognition of the `Result` is complete and the recognizer is confident it has the correct result (not a rejected result). Non-rejection is not a guarantee of a correct result - only sufficient confidence that the guess is correct.
- Events 1: a result transitions from the `UNFINALIZED` state to the `ACCEPTED` state when an `RESULT_ACCEPTED` event is issued.
- Events 2: `AUDIO_RELEASED` and `TRAINING_INFO_RELEASED` events may occur optionally (once) in the `ACCEPTED` state.
- `numTokens` will return 1 or greater (there must be at least one finalized token) and the number of finalized tokens will not change. [Note: A rejected result may have zero finalized tokens.]

- The best guess for each finalized token is available through the `getBestToken(int tokNum)` method. The best guesses will not change through the remaining life of the result.
- `getUnfinalizedTokens` method returns `null`.
- The `getGrammar` method returns the grammar matched by this result. It may be either a `RuleGrammar` or `DictationGrammar`.
- For either a `RuleGrammar` or `DictationGrammar` the methods of `FinalResult` may be used to access audio data and to perform correction/training.
- If the result matches a `RuleGrammar`, the methods of `FinalRuleResult` may be used to get alternative guesses for the complete utterance and to get tags and other information associated with the `RuleGrammar`. (Calls to any methods of the `FinalDictationResult` interface cause a `ResultStateError`.)
- If the result matches a `DictationGrammar`, the methods of `FinalDictationResult` may be used to get alternative guesses for tokens and token sequences. (Calls to any methods of the `FinalRuleResult` interface cause a `ResultStateError`.)

`getResultState() == Result.REJECTED`

- Recognition of the `Result` is complete but the recognizer believes it does not have the correct result. Programmatically, an accepted and rejected result are very similar but the contents of a rejected result must be treated differently - they are likely to be wrong.
- Events 1: a result transitions from the `UNFINALIZED` state to the `REJECTED` state when an `RESULT_REJECTED` event is issued.
- Events 2: (same as for the `ACCEPTED` state) `AUDIO_RELEASED` and `TRAINING_INFO_RELEASED` events may occur optionally (once) in the `REJECTED` state.
- `numTokens` will return 0 or greater. The number of tokens will not change for the remaining life of the result. [Note: an accepted result always has at least one finalized token.]
- As with an accepted result, the best guess for each finalized token is available through the `getBestToken(int num)` method and the tokens are guaranteed not to change through the remaining life of the result. Because the result has been rejected the guesses are not likely to be correct.
- `getUnfinalizedTokens` method returns `null`.
- If the `GRAMMAR_FINALIZED` was issued during recognition of the result, the `getGrammar` method returns the grammar matched by this result otherwise it returns `null`. It may be either a `RuleGrammar` or `DictationGrammar`. For rejected results, there is a greater chance that this grammar is wrong.
- The `FinalResult` interface behaves the same as for a result in the `ACCEPTED` state expect that the information is less likely to be reliable.
- If the grammar is known, the `FinalRuleResult` and `FinalDictationResult` interfaces behave the same as for a result in the `ACCEPTED` state expect that the information is less likely to be reliable. If the grammar is unknown, then a `ResultStateError` is thrown on calls to the methods of both `FinalRuleResult` and `FinalDictationResult`.

Result State and Recognizer States

The state system of a `Recognizer` is linked to the state of recognition of the current result. The `Recognizer` interface documents the normal event cycle for a `Recognizer` and for `Results`. The following is an overview of the ways in which the two state systems are linked:

- The `ALLOCATED` state of a `Recognizer` has three sub-states. In the `LISTENING` state, the recognizer is listening to background audio and there is no result being produced. In the `SUSPENDED` state, the recognizer is temporarily buffering audio input while its grammars are updated. In the `PROCESSING` state, the recognizer has detected incoming audio that may match an active grammar and is producing a `Result`.

- The `Recognizer` moves from the `LISTENING` state to the `PROCESSING` state with a `RECOGNIZER_PROCESSING` event immediately prior to issuing a `RESULT_CREATED` event.
- The `RESULT_UPDATED` and `GRAMMAR_FINALIZED` events are produced while the `Recognizer` is in the `PROCESSING` state.
- The `Recognizer` finalizes a `Result` with `RESULT_ACCEPTED` or `RESULT_REJECTED` event immediately after it transitions from the `PROCESSING` state to the `SUSPENDED` state with a `RECOGNIZER_SUSPENDED` event.
- Unless there is a pending `suspend`, the `Recognizer` commits grammar changes with a `CHANGES_COMMITTED` event as soon as the result finalization event is processed.
- The `TRAINING_INFO_RELEASED` and `AUDIO_RELEASED` events can occur in any state of an `ALLOCATED Recognizer`.

Accept or Reject?

Rejection of a result indicates that the recognizer is not confident that it has accurately recognized what a user says. Rejection can be controlled through the `RecognizerProperties` interface with the `setConfidenceLevel` method. Increasing the confidence level requires the recognizer to have greater confident to accept a result, so more results are likely to be rejected.

Important: the acceptance of a result (an `RESULT_ACCEPTED` event rather than a `RESULT_REJECTED` event) does not mean the result is correct. Instead acceptance implies that the recognizer has a sufficient level of confidence that the result is correct.

It is difficult for recognizers to reliably determine when they make mistakes. Applications need to determine the cost of incorrect recognition of any particular results and take appropriate actions. For example, confirm with a user that they said "delete all files" before deleting anything.

Result Events

Events are issued when a new result is created and when there is any change in the state or information content of a result. The following describes the event sequence for an accepted result. It provides the same information as above for result states, but focusses on legal event sequences.

Before a new result is created for incoming speech, a recognizer usually issues a SPEECH_STARTED event to the speechStarted method of RecognizerAudioListener. The

A newly created `Result` is provided to the application by calling the `resultCreated` method of each `ResultListener` attached to the `Recognizer` with a `RESULT_CREATED` event. The new result may or may not have any finalized tokens or unfinalized tokens.

At any time following the `RESULT_CREATED` event, an application may attach a `ResultListener` to an individual result. That listener will receive all subsequent events associated with that `Result`.

A new `Result` is created in the `UNFINALIZED` state. In this state, zero or more `RESULT_UPDATED` events may be issued to each `ResultListener` attached to the `Recognizer` and to each `ResultListener` attached to that `Result`. The `RESULT_UPDATED` indicates that one or more tokens have been finalized, or that the unfinalized tokens have changed, or both.

When the recognizer determines which grammar is the best match for incoming speech, it issues a `GRAMMAR_FINALIZED` event. This event is issued to each `ResultListener` attached to the `Recognizer` and to each `ResultListener` attached to that `Result`.

The `GRAMMAR_FINALIZED` event is also issued to each `ResultListener` attached to the matched `Grammar`. This is the first `ResultEvent` received by `ResultListeners` attached to the `Grammar`. All subsequent result events are issued to all `ResultListeners` attached to the matched `Grammar` (as well as to `ResultListeners` attached to the `Result` and to the `Recognizer`).

Zero or more `RESULT_UPDATED` events may be issued after the `GRAMMAR_FINALIZED` event but before the result is finalized.

Once the recognizer completes recognition of the `Result` that it choses to accept, it finalizes the result with an `RESULT_ACCEPTED` event that is issued to the `ResultListeners` attached to the `Recognizer`, the matched `Grammar`, and the `Result`. This event may also indicate finalization of zero or more tokens, and/or the reseting of the unfinalized tokens to `null`. The result finalization event occurs immediately after the `Recognizer` makes a transition from the `PROCESSING` state to the `SUSPENDED` state with a `RECOGNIZER_SUSPENDED` event.

A finalized result (accepted or rejected state) may issue a `AUDIO_RELEASED` or `TRAINING_INFO_RELEASED` event. These events may be issued in response to relevant release methods of `FinalResult` and `FinalDictationResult` or may be issued when the recognizer independently determines to release audio or training information.

When a result is rejected some of the events described above may be skipped. A result may be rejected with the `RESULT_REJECTED` event at any time after a `RESULT_CREATED` event instead of an `RESULT_ACCEPTED` event. A result may be rejected with or without any unfinalized or finalized tokens being created (no `RESULT_UPDATED` events), and with or without a `GRAMMAR_FINALIZED` event.

When does a `Result` start and end?

A new result object is created when a recognizer has detected possible incoming speech which may match an active grammar.

To accept the result (i.e. to issue a `RESULT_ACCEPTED` event), the best-guess tokens of the result must match the token patterns defined by the matched grammar. For a `RuleGrammar` this implies that a call to the `parse` method of the matched `RuleGrammar` must return successfully. (Note: the parse is not guaranteed if the grammar has been changed.)

Because there are no programmatically defined constraints upon word patterns for a `DictationGrammar`, a single result may represent a single word, a short phrase or sentence, or possibly many pages of text.

The set of conditions that may cause a result matching a `DictationGrammar` to be finalized includes:

- The user pauses for a period of time (a timeout).
- A call to the forceFinalize method of the recognizer.
- User has spoken text matching an active `RuleGrammar` (the dictation result is finalized and a new `Result` is issued for the `RuleGrammar`).
- The engine is paused.

The following conditions apply to all finalized results:

- N-best alternative token guesses available through the `FinalRuleResult` and `FinalDictationResult` interfaces cannot cross result boundaries.
- Correction/training is only possible within a single result object.

See Also:
FinalResult, FinalRuleResult, FinalDictationResult, ResultEvent, ResultListener, ResultAdapter, Grammar, RuleGrammar, DictationGrammar, forceFinalize, RecognizerEvent, setConfidenceLevel

---

Field Summary

static int	**ACCEPTED** `getResultState` returns `ACCEPTED` once recognition of the result is completed and the `Result` object has been finalized by being accepted.
static int	**REJECTED** `getResultState` returns `REJECTED` once recognition of the result complete and the `Result` object has been finalized by being rejected.
static int	**UNFINALIZED** `getResultState` returns `UNFINALIZED` while a result is still being recognized.

Method Summary

void	**addResultListener**(ResultListener listener) Request notifications of events of related to this `Result`.
ResultToken	**getBestToken**(int tokNum) Returns the best guess for the `tokNum` th token.
ResultToken[]	**getBestTokens**() Returns all the best guess tokens for this result.
Grammar	**getGrammar**() Return the `Grammar` matched by the best-guess finalized tokens of this result or `null` if the grammar is not known.
int	**getResultState**() Returns the current state of the Result object: `UNFINALIZED`, `ACCEPTED` or `REJECTED`.
ResultToken[]	**getUnfinalizedTokens**() In the `UNFINALIZED` state, return the current guess of the tokens following the finalized tokens.
int	**numTokens**() Returns the number of finalized tokens in a `Result`.
void	**removeResultListener**(ResultListener listener) Remove a listener from this `Result`.

Field Detail

UNFINALIZED

`public static final int` **`UNFINALIZED`**

`getResultState` returns `UNFINALIZED` while a result is still being recognized. A `Result` is in the `UNFINALIZED` state when the `RESULT_CREATED` event is issued. Result states are described above in detail.

See Also:
getResultState, RESULT_CREATED

ACCEPTED

`public static final int` **`ACCEPTED`**

`getResultState` returns `ACCEPTED` once recognition of the result is completed and the `Result` object has been finalized by being accepted. When a `Result` changes to the `ACCEPTED` state a `RESULT_ACCEPTED` event is issued. Result states are described above in detail.

See Also:
getResultState, RESULT_ACCEPTED

REJECTED

`public static final int` **`REJECTED`**

`getResultState` returns `REJECTED` once recognition of the result complete and the `Result` object has been finalized by being rejected. When a `Result` changes to the `REJECTED` state a `RESULT_REJECTED` event is issued. Result states are described above in detail.

See Also:
getResultState, RESULT_REJECTED

Method Detail

getResultState

`public int` **`getResultState`**`()`

Returns the current state of the Result object: `UNFINALIZED`, `ACCEPTED` or `REJECTED`. The details of a `Result` in each state are described above.

See Also:
UNFINALIZED, ACCEPTED, REJECTED

---

getGrammar

`public Grammar **getGrammar**()`

Return the `Grammar` matched by the best-guess finalized tokens of this result or `null` if the grammar is not known. The return value is `null` before a `GRAMMAR_FINALIZED` event and non-null afterwards.

The grammar is guaranteed to be non-null for an accepted result. The grammar may be null or non-null for a rejected result, depending upon whether a `GRAMMAR_FINALIZED` event was issued prior to finalization.

For a finalized result, an application should determine the type of matched grammar with an `instanceof` test. For a result that matches a `RuleGrammar`, the methods of `FinalRuleResult` can be used (the methods of `FinalDictationResult` throw an error). For a result that matches a `DictationGrammar`, the methods of `FinalDictationResult` can be used (the methods of `FinalRuleResult` throw an error). The methods of `FinalResult` can be used for a result matching either kind of grammar.

Example:

```
Result result;
if (result.getGrammar() instanceof RuleGrammar) {
  FinalRuleResult frr = (FinalRuleResult)result;
  ...
}
```

See Also:
getResultState

---

numTokens

`public int **numTokens**()`

Returns the number of finalized tokens in a `Result`. Tokens are numbered from `0` to `numTokens()-1` and are obtained through the `getBestToken` and `getBestTokens` method of this (`Result`) interface and the `getAlternativeTokens` methods of the `FinalRuleResult` and `FinalDictationResult` interfaces for a finalized result.

Starting from the `RESULT_CREATED` event and while the result remains in the `UNFINALIZED` state, the number of finalized tokens may be zero or greater and can increase as tokens are finalized. When one or more tokens are finalized in the `UNFINALIZED` state, a `RESULT_UPDATED` event is issued with the `tokenFinalized` flag set `true`. The `RESULT_ACCEPTED` and `RESULT_REJECTED` events which finalize a result can also indicate that one or more tokens have been finalized.

In the `ACCEPTED` and `REJECTED` states, `numTokens` indicates the total number of tokens that were finalized. The number of finalized tokens never changes in these states. An `ACCEPTED` result must have one or more finalized token. A `REJECTED` result may have zero or more tokens.

See Also:
RESULT_UPDATED, getBestToken, getBestTokens, getAlternativeTokens, getAlternativeTokens

---

getBestToken

```
public ResultToken getBestToken(int tokNum)
                                         throws java.lang.IllegalArgumentException
```

Returns the best guess for the `tokNum`th token. `tokNum` must be in the range `0` to `numTokens()-1`.

If the result has zero tokens (possible in both the `UNFINALIZED` and `REJECTED` states) an exception is thrown.

If the result is in the `REJECTED` state, then the returned tokens are likely to be incorrect. In the `ACCEPTED` state (not rejected) the recognizer is confident that the tokens are correct but applications should consider the possibility that the tokens are incorrect.

The `FinalRuleResult` and `FinalDictationResult` interfaces provide `getAlternativeTokens` methods that return alternative token guesses for finalized results.

Throws:
java.lang.IllegalArgumentException - if `tokNum` is out of range.
See Also:
getUnfinalizedTokens, getBestTokens, getAlternativeTokens, getAlternativeTokens

---

getBestTokens

`public ResultToken[] **getBestTokens**()`

Returns all the best guess tokens for this result. If the result has zero tokens, the return value is `null`.

---

getUnfinalizedTokens

`public ResultToken[] **getUnfinalizedTokens**()`

In the `UNFINALIZED` state, return the current guess of the tokens following the finalized tokens. Unfinalized tokens provide an indication of what a recognizer is considering as possible recognition tokens for speech following the finalized tokens.

. * Unfinalized tokens can provide users with feedback on the recognition process. The array may be any length (zero or more tokens), the length may change at any time, and successive calls to `getUnfinalizedTokens` may return different tokens or even different numbers of tokens. When the unfinalized tokens are changed, a `RESULT_UPDATED` event is issued to the `ResultListener`. The `RESULT_ACCEPTED` and `RESULT_REJECTED` events finalize a result and always guarantee that the return value is `null`. A new result created with a `RESULT_CREATED` event may have a null or non-null value.

The returned array is null if there are currently no unfinalized tokens, if the recognizer does not support unfinalized tokens, or after a `Result` is finalized (in the `ACCEPTED` or `REJECTED` state).

See Also:
isUnfinalizedTokensChanged, RESULT_UPDATED, RESULT_ACCEPTED, RESULT_REJECTED

---

addResultListener

`public void addResultListener(ResultListener listener)`

Request notifications of events of related to this `Result`. An application can attach multiple listeners to a `Result`. A listener can be removed with the `removeResultListener` method.

`ResultListener` objects can also be attached to a `Recognizer` and to any `Grammar`. A listener attached to the `Recognizer` receives all events for all results produced by that `Recognizer`. A listener attached to a `Grammar` receives all events for all results that have been finalized for that `Grammar` (all events starting with and including the `GRAMMAR_FINALIZED` event).

A `ResultListener` attached to a `Result` only receives events following the point in time at which the listener is attached. Because the listener can only be attached during or after the `RESULT_CREATED`, it will not receive the `RESULT_CREATED` event. Only `ResultListeners` attached to the `Recognizer` receive `RESULT_CREATED` events.

See Also:
removeResultListener, addResultListener, addResultListener

removeResultListener

`public void removeResultListener(ResultListener listener)`

Remove a listener from this `Result`.

See Also:
addResultListener, removeResultListener, removeResultListener

Class javax.speech.recognition.ResultAdapter

```
java.lang.Object
  |
  +--javax.speech.recognition.ResultAdapter
```

public class **ResultAdapter**
extends java.lang.Object
implements ResultListener

The adapter which receives events for a `Result` object. The methods in this class are empty; this class is provided as a convenience for easily creating listeners by extending this class and overriding only the methods of interest.

Method Summary

void	**audioReleased**(ResultEvent e) A `AUDIO_RELEASED` event has occured.
void	**grammarFinalized**(ResultEvent e) A `GRAMMAR_FINALIZED` event has occured.
void	**resultAccepted**(ResultEvent e) A `RESULT_ACCEPTED` event has occured.
void	**resultCreated**(ResultEvent e) A `RESULT_CREATED` event has occured.
void	**resultRejected**(ResultEvent e) A `RESULT_REJECTED` event has occured.
void	**resultUpdated**(ResultEvent e) A `RESULT_UPDATED` event has occured.
void	**trainingInfoReleased**(ResultEvent e) A `TRAINING_INFO_RELEASED` event has occured.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Method Detail

resultCreated

`public void `**`resultCreated`**`(ResultEvent e)`

A `RESULT_CREATED` event has occured.

Specified by:
resultCreated in interface ResultListener
See Also:
RESULT_CREATED

---

resultUpdated

`public void `**`resultUpdated`**`(ResultEvent e)`

A `RESULT_UPDATED` event has occured.

Specified by:
resultUpdated in interface ResultListener
See Also:
RESULT_UPDATED

---

grammarFinalized

`public void `**`grammarFinalized`**`(ResultEvent e)`

A `GRAMMAR_FINALIZED` event has occured.

Specified by:
grammarFinalized in interface ResultListener
See Also:
GRAMMAR_FINALIZED

---

resultAccepted

`public void `**`resultAccepted`**`(ResultEvent e)`

A `RESULT_ACCEPTED` event has occured.

Specified by:
resultAccepted in interface ResultListener
See Also:
RESULT_ACCEPTED

---

resultRejected

`public void `**`resultRejected`**`(ResultEvent e)`

A `RESULT_REJECTED` event has occured.

Specified by:
resultRejected in interface ResultListener
See Also:
RESULT_REJECTED

---

audioReleased

`public void `**`audioReleased`**`(ResultEvent e)`

A `AUDIO_RELEASED` event has occured.

Specified by:
audioReleased in interface ResultListener
See Also:
AUDIO_RELEASED

---

trainingInfoReleased

`public void `**`trainingInfoReleased`**`(ResultEvent e)`

A `TRAINING_INFO_RELEASED` event has occured.

Specified by:
trainingInfoReleased in interface ResultListener
See Also:
TRAINING_INFO_RELEASED

Class javax.speech.recognition.ResultEvent

```
java.lang.Object
  |
  +--java.util.EventObject
        |
        +--javax.speech.SpeechEvent
              |
              +--javax.speech.recognition.ResultEvent
```

public class **ResultEvent**
extends SpeechEvent

A `ResultEvent` is issued by a `Result` object to indicate changes in the recognized tokens and changes in state. An event is issued to all appropriate `ResultListener` objects. `ResultListeners` can be attached in three places:

- A `ResultListener` attached to the `Recognizer` receives all events for all results produced by the recognizer.
- A `ResultListener` attached to a `Grammar` receives all events for all results that have been finalized for that `Grammar`: the `GRAMMAR_FINALIZED` event and following events. (Note: it never receives a `RESULT_CREATED` event because that event always precedes `GRAMMAR_FINALIZED`).
- A `ResultListener` attached to a `Result` receives all events for that result starting from the time at which the listener is attached. (Note: it never receives a `RESULT_CREATED` event because a listener can only be attached to a `Result` once a `RESULT_CREATED` event has already been issued.)

In all three forms of listener attachment, one listener can be attached to multiple objects, and multiple listeners can be attached to a single object.

The three states of a recognition result are described in the documentation for the `Result` interface and can be tested by the `getResultState` method of that interface. The `RESULT_CREATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events indicate result state changes.

In brief, the three states are:

- `UNFINALIZED` state: initial state of result indicating that recognition is still taking place. A new result is created in the `UNFINALIZED` state with a `RESULT_CREATED` event.
- `ACCEPTED` state: the result is finalized and the recognizer is confident it has recognized the result correctly. The `RESULT_ACCEPTED` event indicates a state change from the `UNFINALIZED` state to the `ACCEPTED` state.
- `REJECTED` state: the result is finalized but the recognizer is not confident it has recognized the result correctly. The `RESULT_REJECTED` event indicates a state change from the `UNFINALIZED` state to the `REJECTED` state.

The sequence of `ResultEvents` associated with a recognition result are described in the documentation for the `Result` interface.

In brief, the events that occur depend upon the `Result` state:

- A `RESULT_CREATED` event creates a `Result` object. A new result is starts in the `UNFINALIZED` state.
- `UNFINALIZED` state: `RESULT_UPDATED` events indicate a change in finalized and/or unfinalized tokens; a `GRAMMAR_FINALIZED` event indicates that the `Grammar` matched by this result has been identified.
- The `RESULT_ACCEPTED` event finalizes a result by indicating a change in state from `UNFINALIZED` to `ACCEPTED`.
- The `RESULT_REJECTED` event finalizes a result by indicating a change in state from `UNFINALIZED` to `REJECTED`.
- In the finalized states - `ACCEPTED` and `REJECTED` - the `AUDIO_RELEASED` and `TRAINING_INFO_RELEASED` may be issued.

See Also:
Recognizer, Result, ResultListener, FinalResult, FinalRuleResult, FinalDictationResult, Serialized Form

Field Summary

static int	**AUDIO_RELEASED** `AUDIO_RELEASED` event is issued when the audio information associated with a `FinalResult` object is released.
static int	**GRAMMAR_FINALIZED** `GRAMMAR_FINALIZED` is issued when the `Grammar` matched by a `Result` is identified and finalized.
static int	**RESULT_ACCEPTED** `RESULT_ACCEPTED` event is issued when a `Result` is successfully finalized and indicates a state change from `UNFINALIZED` to `ACCEPTED`.
static int	**RESULT_CREATED** `RESULT_CREATED` is issued when a new `Result` is created.
static int	**RESULT_REJECTED** `RESULT_REJECTED` event is issued when a `Result` is unsuccessfully finalized and indicates a change from the `UNFINALIZED` state to the `REJECTED` state.
static int	**RESULT_UPDATED** `RESULT_UPDATED` is issued when one or more tokens of a `Result` are finalized or when the unfinalized tokens of a result are changed.
boolean	**tokenFinalized** True if the `ResultEvent` indicates that one or more tokens have been finalized.
static int	**TRAINING_INFO_RELEASED** `TRAINING_INFO_RELEASED` event is issued when the training information for a finaliized result is released.
boolean	**unfinalizedTokensChanged** True if the `ResultEvent` indicates that the unfinalized tokens have changed.

Fields inherited from class javax.speech.SpeechEvent
id

Fields inherited from class java.util.EventObject
source

Constructor Summary
ResultEvent(Result source, int id, boolean isTokenFinalized, boolean isUnfinalizedTokensChanged) Constructs a `ResultEvent` for a specified source `Result` and result event id.
ResultEvent(Result source, int id) Constructs a `ResultEvent` with an event type identifier.

Method Summary	
boolean	**isTokenFinalized**() For `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events returns `true` if any tokens were finalized.
boolean	**isUnfinalizedTokensChanged**() For `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events returns `true` if the unfinalized tokens changed.
java.lang.String	**paramString**() Returns a parameter string identifying this event.

Methods inherited from class javax.speech.SpeechEvent
getId, paramString, toString

Methods inherited from class java.util.EventObject
getSource, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

RESULT_CREATED

`public static final int` **`RESULT_CREATED`**

`RESULT_CREATED` is issued when a new `Result` is created. The event is received by each `ResultListener` attached to the `Recognizer`.

When a result is created, it is in the `UNFINALIZED` state. When created the result may have zero or more finalized tokens and zero or more unfinalized tokens. The presence of finalized and unfinalized tokens is indicated by the `isTokenFinalized` and `isUnfinalizedTokensChanged` flags.

The `RESULT_CREATED` event follows the `RECOGNIZER_PROCESSING` event which transitions the `Recognizer` from the `LISTENING` state to the `PROCESSING` state.

See Also:
resultCreated, UNFINALIZED, PROCESSING, RECOGNIZER_PROCESSING, isTokenFinalized, isUnfinalizedTokensChanged

---

RESULT_UPDATED

`public static final int` **`RESULT_UPDATED`**

`RESULT_UPDATED` is issued when one or more tokens of a `Result` are finalized or when the unfinalized tokens of a result are changed. The `isTokenFinalized` and `isUnfinalizedTokensChanged` flags are set appropriately.

The `RESULT_UPDATED` event only occurs when a `Result` is in the `UNFINALIZED` state.

See Also:
isTokenFinalized, isUnfinalizedTokensChanged, UNFINALIZED, resultUpdated

---

GRAMMAR_FINALIZED

`public static final int` **`GRAMMAR_FINALIZED`**

`GRAMMAR_FINALIZED` is issued when the `Grammar` matched by a `Result` is identified and finalized. Before this event the `getGrammar` method of a `Result` returns `null`. Following the event it is guaranteed to return non-null and the grammar is guaranteed not to change.

The `GRAMMAR_FINALIZED` event only occurs for a `Result` is in the `UNFINALIZED` state.

A `GRAMMAR_FINALIZED` event does not affect finalized or unfinalized tokens.

See Also:
isTokenFinalized, isUnfinalizedTokensChanged, UNFINALIZED, grammarFinalized

---

RESULT_ACCEPTED

`public static final int` **`RESULT_ACCEPTED`**

`RESULT_ACCEPTED` event is issued when a `Result` is successfully finalized and indicates a state change from `UNFINALIZED` to `ACCEPTED`.

In the finalization transition, zero or more tokens may be finalized and the unfinalized tokens are set to `null`. The `isTokenFinalized` and `isUnfinalizedTokensChanged` flags are set appropriately.

Since the `Result` is finalized (accepted), the methods of `FinalResult` and either `FinalRuleResult` or `FinalDictationResult` can be used. (Use the `getGrammar` method of `Result` to determine the type of grammar matched.) Applications should use type casting to ensure that only the appropriate interfaces and method are used.

The `RESULT_ACCEPTED` event is issued after the `Recognizer` issues a `RECOGNIZER_SUSPENDED` event to transition from the `PROCESSING` state to the `SUSPENDED` state. Any changes made to grammars or the enabled state of grammars during the processing of the `RESULT_ACCEPTED` event are automatically committed once the `RESULT_ACCEPTED` event has been processed by all `ResultListeners`. Once those changes have been committed, the `Recognizer` returns to the `LISTENING` state with a `CHANGES_COMMITTED` event. A call to `commitChanges` is not required. (Except, if there is a call to `suspend` without a subsequent call to `commitChanges`, the `Recognizer` defers the commit until the `commitChanges` call is received.)

See Also:
resultAccepted, isTokenFinalized, isUnfinalizedTokensChanged, getResultState, getGrammar, SUSPENDED, RECOGNIZER_SUSPENDED

RESULT_REJECTED

`public static final int` **`RESULT_REJECTED`**

`RESULT_REJECTED` event is issued when a `Result` is unsuccessfully finalized and indicates a change from the `UNFINALIZED` state to the `REJECTED` state.

In the state transition, zero or more tokens may be finalized and the unfinalized tokens are set to `null`. The `isTokenFinalized` and `isUnfinalizedTokensChanged` flags are set appropriately. However, because the result is rejected, the tokens are quite likely to be incorrect.

Since the `Result` is finalized (rejected), the methods of `FinalResult` can be used. If the grammar is known (`GRAMMAR_FINALIZED` event was issued and the `getGrammar` methods returns non-null) then the `FinalRuleResult` or `FinalDictationResult` interface can also be used depending upon whether the matched grammar was a `RuleGrammar` or `DictationGrammar`.

Other state transition behavior for `RESULT_REJECTED` is the same as for the `RESULT_ACCEPTED` event.

See Also:
isTokenFinalized, isUnfinalizedTokensChanged, getResultState, getGrammar, resultRejected

---

AUDIO_RELEASED

`public static final int` **`AUDIO_RELEASED`**

`AUDIO_RELEASED` event is issued when the audio information associated with a `FinalResult` object is released. The release may have been requested by an application call to `releaseAudio` in the `FinalResult` interface or may be initiated by the recognizer to reclaim memory. The `FinalResult.isAudioAvailable` method returns `false` after this event.

The `AUDIO_RELEASED` event is only issued for results in a finalized state (`getResultState` returns either `ACCEPTED` or `REJECTED`).

See Also:
releaseAudio, isAudioAvailable, getResultState, audioReleased

---

TRAINING_INFO_RELEASED

`public static final int` **`TRAINING_INFO_RELEASED`**

`TRAINING_INFO_RELEASED` event is issued when the training information for a finaliized result is released. The release may have been requested by an application call to the `releaseTrainingInfo` method in the `FinalResult` interface or may be initiated by the recognizer to reclaim memory. The `isTrainingInfoAvailable` method of `FinalResult` returns `false` after this event.

The `TRAINING_INFO_RELEASED` event is only issued for results in a finalized state (`getResultState` returns either `ACCEPTED` or `REJECTED`).

See Also:
releaseTrainingInfo, isTrainingInfoAvailable, getResultState, trainingInfoReleased

---

tokenFinalized

`protected boolean` **`tokenFinalized`**

True if the `ResultEvent` indicates that one or more tokens have been finalized. Tokens may be finalized with any of the following result events: `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED`, `RESULT_REJECTED`.

See Also:
isTokenFinalized

---

unfinalizedTokensChanged

`protected boolean` **`unfinalizedTokensChanged`**

True if the `ResultEvent` indicates that the unfinalized tokens have changed. The unfinalized tokens may change with the following result events: `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED`, `RESULT_REJECTED`.

See Also:
isUnfinalizedTokensChanged

Constructor Detail

ResultEvent

```
public ResultEvent(Result source,
                   int id,
                   boolean isTokenFinalized,
                   boolean isUnfinalizedTokensChanged)
```

Constructs a `ResultEvent` for a specified source `Result` and result event id. The two boolean flags indicating change in tokens should be set appropriately for `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events. (For other event types these flags should be `false`).

Parameters:
`source` - the `Result` object that issued the event
`id` - the identifier for the event type
`isTokenFinalized` - true if any token is finalized with this event
`isUnfinalizedTokensChanged` - true if the unfinalized text is changed with this event
See Also:
RESULT_CREATED, GRAMMAR_FINALIZED, RESULT_UPDATED, RESULT_ACCEPTED, RESULT_REJECTED, AUDIO_RELEASED, TRAINING_INFO_RELEASED

ResultEvent

```
public ResultEvent(Result source,
                   int id)
```

Constructs a `ResultEvent` with an event type identifier. The `isTokenFinalized` and `isUnfinalizedTokensChanged` flags are set to `false`.

Parameters:
`source` - the object that issued the event
`id` - the identifier for the event type

Method Detail

isTokenFinalized

`public boolean` **`isTokenFinalized`**`()`

For `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events returns `true` if any tokens were finalized. For other events, return `false`. If true, the number of tokens returned by `numTokens` and `getBestTokens` has increased.

See Also:
numTokens, getBestTokens

---

isUnfinalizedTokensChanged

`public boolean` **`isUnfinalizedTokensChanged`**`()`

For `RESULT_CREATED`, `RESULT_UPDATED`, `RESULT_ACCEPTED` and `RESULT_REJECTED` events returns `true` if the unfinalized tokens changed. For other events, return `false`. If true, the value returned by `getUnfinalizedTokens` has changed.

Note that both `RESULT_ACCEPTED` and `RESULT_REJECTED` events implicitly set the unfinalized text to `null`. The `isUnfinalizedTokensChanged` method should return `true` only if the unfinalized text was non-null prior to finalization.

See Also:
getUnfinalizedTokens

---

paramString

`public java.lang.String` **`paramString`**`()`

Returns a parameter string identifying this event. This method is useful for event-logging and for debugging.

Returns:
a string identifying the event
Overrides:
paramString in class SpeechEvent

Interface javax.speech.recognition.ResultListener

Implementing Classes:
ResultAdapter

---

public abstract interface **ResultListener**
extends java.util.EventListener

The methods of a `ResultListener` receive notifications of events related to a `Result` object. A `ResultListener` may be attached to any of three entities:

- **`Recognizer`**: a `ResultListener` attached to a `Recognizer` receives notification of all `ResultEvents` for all results produced by that recognizer.
- **`Grammar`**: a `ResultListener` attached to a `Grammar` receives all events for all results that have been finalized for that grammar. Specifically, it receives the `GRAMMAR_FINALIZED` event and following events. (Note: it never receives a `RESULT_CREATED` event because that event always precedes the `GRAMMAR_FINALIZED` event).
- **`Result`**: a `ResultListener` attached to a `Result` receives all events for that result starting from the time at which the listener is attached. (Note: it never receives a `RESULT_CREATED` event because a listener can only be attached to a `Result` once a `RESULT_CREATED` event has already been issued.)

The events are issued to the listeners in the order of most specific to least specific: `ResultListeners` attached to the `Result` are notified first, then listeners attached to the matched `Grammar`, and finally to listeners attached to the `Recognizer`.

A single `ResultListener` may be attached to multiple objects (`Recognizer`, `Grammar` or `Result`), and multiple `ResultListeners` may be attached to a single object.

The source for all `ResultEvents` issued to a `ResultListener` is the `Result` that generated the event. The full state system for results and the associated events are described in the documentation for `ResultEvent`.

A trivial implementation of the `ResultListener` interface is provided by the `ResultAdapter` class.

See Also:
ResultEvent, ResultAdapter, addResultListener(ResultListener), addResultListener(ResultListener), addResultListener(ResultListener)

Method Summary

void	**audioReleased**(ResultEvent e) A `AUDIO_RELEASED` event has occured.
void	**grammarFinalized**(ResultEvent e) A `GRAMMAR_FINALIZED` event has occured because the `Recognizer` has determined which `Grammar` is matched by the incoming speech.
void	**resultAccepted**(ResultEvent e) An `RESULT_ACCEPTED` event has occured indicating that a `Result` has transitioned from the `UNFINALIZED` state to the `ACCEPTED` state.
void	**resultCreated**(ResultEvent e) A `RESULT_CREATED` event is issued when a `Recognizer` detects incoming speech that may match an active grammar of an application.
void	**resultRejected**(ResultEvent e) An `RESULT_REJECTED` event has occured indicating that a `Result` has transitioned from the `UNFINALIZED` state to the `REJECTED` state.
void	**resultUpdated**(ResultEvent e) A `RESULT_UPDATED` event has occured because a token has been finalized and/or the unfinalized text of a result has changed.
void	**trainingInfoReleased**(ResultEvent e) A `TRAINING_INFO_RELEASED` event has occured.

Method Detail

resultCreated

`public void resultCreated(ResultEvent e)`

A `RESULT_CREATED` event is issued when a `Recognizer` detects incoming speech that may match an active grammar of an application.

The event is issued to each `ResultListener` attached to the `Recognizer`. (`ResultListeners` attached to a `Grammar` or to a `Result` never receive a `RESULT_CREATED` event.)

The `RESULT_CREATED` follows the `RECOGNIZER_PROCESSING` event that is issued `RecognizerListeners` to indicate that the `Recognizer` has changed from the `LISTENING` to the `PROCESSING` state.

See Also:
RESULT_CREATED, RECOGNIZER_PROCESSING

---

resultUpdated

`public void` **`resultUpdated`**`(ResultEvent e)`

A `RESULT_UPDATED` event has occured because a token has been finalized and/or the unfinalized text of a result has changed.

The event is issued to each `ResultListener` attached to the `Recognizer`, to each `ResultListener` attached to the `Result`, and if the `GRAMMAR_FINALIZED` event has already been released to each `ResultListener` attached to the matched `Grammar`.

See Also:
RESULT_UPDATED

---

grammarFinalized

`public void` **`grammarFinalized`**`(ResultEvent e)`

A `GRAMMAR_FINALIZED` event has occured because the `Recognizer` has determined which `Grammar` is matched by the incoming speech.

The event is issued to each `ResultListener` attached to the `Recognizer`, `Result`, and matched `Grammar`.

See Also:
GRAMMAR_FINALIZED

---

resultAccepted

`public void` **`resultAccepted`**`(ResultEvent e)`

An `RESULT_ACCEPTED` event has occured indicating that a `Result` has transitioned from the `UNFINALIZED` state to the `ACCEPTED` state.

Since the `Result` source for this event is finalized, the `Result` object can be safely cast to the `FinalResult` interface.

Because the result is accepted, the matched `Grammar` for the result is guaranteed to be non-null. If the matched `Grammar` is a `RuleGrammar`, then the result object can be safely cast to `FinalRuleResult` (methods of `FinalDictationResult` throw an exception). If the matched `Grammar` is a `DictationGrammar`, then the result object can be safely cast to `FinalDictationResult` (methods of `FinalRuleResult` throw an exception). For example:

```
void resultAccepted(ResultEvent e) {
   Object source = e.getSource();

   // always safe
   Result result = (Result)source;
   FinalResult fr = (FinalResult)source;

   // find the grammar-specific details
   // the null test is only useful for rejected results
   if (result.getGrammar() != null) {
     if (result.getGrammar() instanceof RuleGrammar) {
       FinalRuleResult frr = (FinalRuleResult)source;
       ...
     }
     else if (result.getGrammar() instanceof DictationGrammar) {
       FinalDictationResult fdr = (FinalDictationResult)source;
       ...
     }
```

The event is issued to each `ResultListener` attached to the `Recognizer`, `Result`, and matched `Grammar`.

The `RESULT_ACCEPTED` event is issued following the `RECOGNIZER_SUSPENDED` `RecognizerEvent` and while the `Recognizer` is in the `SUSPENDED` state. Once the `RESULT_ACCEPTED` event has been processed by all listeners, the `Recognizer` automatically commits all changes to grammars and returns to the `LISTENING` state. The only exception is when a call has been made to `suspend` without a following call to `commitChanges`. In this case the `Recognizer` remains `SUSPENDED` until `commitChanges` is called.

See Also:
RESULT_ACCEPTED, FinalResult, FinalRuleResult, FinalDictationResult, RecognizerEvent, commitChanges

---

resultRejected

`public void` **`resultRejected`**`(ResultEvent e)`

An `RESULT_REJECTED` event has occured indicating that a `Result` has transitioned from the `UNFINALIZED` state to the `REJECTED` state.

The casting behavior of a rejected result is the same as for a `RESULT_ACCEPTED` event. The exception is that if the grammar is not known (no `GRAMMAR_FINALIZED` event), then the result cannot be cast to either `FinalRuleResult` or `FinalDictationResult`.

The state behavior and grammar committing actions are the same as for the `RESULT_ACCEPTED` event.

The event is issued to each `ResultListener` attached to the `Recognizer` and to the `Result`. If a `GRAMMAR_FINALIZED` event was issued, then the matched `Grammar` is known, and the event is also issued to each `ResultListener` attached to that `Grammar`.

See Also:
RESULT_REJECTED, FinalResult, FinalRuleResult, FinalDictationResult

---

audioReleased

`public void` **`audioReleased`**`(ResultEvent e)`

A `AUDIO_RELEASED` event has occured. This event is only issued to finalized results. See the documentation of the `isAudioAvailable` method the `FinalResult` interface for details.

The event is issued to each `ResultListener` attached to the `Recognizer` and to the `Result`. If a `GRAMMAR_FINALIZED` event was issued, then the matched `Grammar` is known, and the event is also issued to each `ResultListener` attached to that `Grammar`.

See Also:
AUDIO_RELEASED, isAudioAvailable

---

trainingInfoReleased

```
public void trainingInfoReleased(ResultEvent e)
```

A `TRAINING_INFO_RELEASED` event has occured. This event is only issued to finalized results. See the documentation of the `isTrainingInfoAvailable` method the `FinalResult` interface for details.

The event is issued to each `ResultListener` attached to the `Recognizer` and to the `Result`. If a `GRAMMAR_FINALIZED` event was issued, then the matched `Grammar` is known, and the event is also issued to each `ResultListener` attached to that `Grammar`.

See Also:
TRAINING_INFO_RELEASED, isTrainingInfoAvailable

Class javax.speech.recognition.ResultStateError

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Error
              |
              +--javax.speech.SpeechError
                    |
                    +--javax.speech.recognition.ResultStateError
```

public class **ResultStateError**
extends SpeechError

Signals an error caused by an illegal call to a method of `FinalResult`, `FinalRuleResult` or `FinalDictationResult`.

Methods of these three interfaces of a result can only be called for a finalized `Result`. The `getResultState` method of the `Result` interface tests the result state and must return either `ACCEPTED` or `REJECTED`. Furthermore, the methods of `FinalRuleResult` should only be called for a finalized result matching a `RuleGrammar`. Similarly, the methods of `FinalDictationResult` should only be called for a finalized result matching a `DictationGrammar`.

See Also:
Serialized Form

Constructor Summary
ResultStateError(java.lang.String s) Construct a `ResultStateError` with a message string.
ResultStateError() Empty constructor for `ResultStateError`.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

ResultStateError

`public ResultStateError(java.lang.String s)`

Construct a `ResultStateError` with a message string.

ResultStateError

`public ResultStateError()`

Empty constructor for `ResultStateError`.

Interface javax.speech.recognition.ResultToken

public abstract interface **ResultToken**

A token (usually a word) contained by a `Result` representing something heard by a recognizer. For a result of a `RuleGrammar` a `ResultToken` contains the same string as the defining `RuleToken` in the `RuleGrammar`. For a result of a `DictationGrammar`, the tokens are defined by the recognizer.

For any result, best guess finalized tokens are obtained from the `getBestToken` and `getBestTokens` methods of the `Result` interface. For a finalized result matching a `RuleGrammar` or a finalized result matching a `DictationGrammar` alternative guesses are available through the `getAlternativeTokens` methods of the `FinalRuleResult` and `FinalDictationResult` interfaces respectively.

The `ResultToken` provides the following information:

- Required: Spoken-form text
- Required: reference to the `Result` that contains this token
- Optional: Start and end time
- Dictation only: Written-form text
- Dictation only: Presentation hints (capitalization and spacing)

Spoken vs. Written Form

The distinction between spoken and written forms of a `ResultToken` applies only to results for a `DictationGrammar`. For a result matching a `RuleGrammar`, the written and spoken forms are identical.

The *spoken form* is a printable string that indicates what the recognizer heard the user say. In a dictation application, the spoken form is typically used when displaying unfinalized tokens.

The *written form* is a printable string that indicates how the recognizer thinks the token should be printed in text. In a dictation application, the written form is typically used when displaying finalized tokens.

For example, in English, a recognizer might return "twenty" for the spoken form and "20" for the written form.

Recognizers perform the conversion from spoken to written form on a per-token basis. For example, "nineteen thirty eight" is three tokens. The written form would also be three tokens: "19 30 8". Applications may use additional processing to convert such token sequences to "1938".

For some `ResultTokens`, the written form is a single Unicode character. Amongst these are the following important formatting characters (shown here as spoken form for US English, written form as a Unicode character number):

- New line character is "\u000A" and equals the static string `NEW_LINE`.
  In English, it might be spoken as "new line", "line break" or similar.
- New paragraph character is "\u2029" and equals the static string `NEW_PARAGRAPH`.

In English, it might be spoken as "new paragraph", "start paragraph" or something similar.

The following are examples of punctuation characters given with sample spoken forms and the corresponding written form. The spoken form may vary between recognizers and one recognizer may even allow one punctuation character to be spoken multiple ways. Also the set of characters may be engine-specific and language-specific.

- "space bar" -> " " (u0020)
- "exclamation mark", "exclamation point" -> "!" (u0021)
- "open quote", "begin quote", "open-\"" -> "\"" (u0022) (single quote char)
- "dash", "hyphen", "minus" -> "-" (u002D)
- "pound sign" -> "£" (u00A3)
- "yen sign" -> "¥" (u00A5)

Presentation Hints

Note: results for rule grammars do not provide presentation hints. Default values are returns for both `SpacingHint` and `CapitalizationHint`.

Applications use the presentation hints as a guide to rendering the written-form tokens into complete strings. The two hints are `SpacingHint` and `CapitalizationHint`.

`SpacingHint` is an `int` with several flags indicating how the token should be spaced.

- `SpacingHint==SEPARATE` (value of 0) when all the flags are false. The token should be surrounding by preceding and following spaces.
- `ATTACH_PREVIOUS`: Flag is `true` if the token should be attached to the previous token: i.e. no space between this token and the previous token.
- `ATTACH_FOLLOWING`: Flag is `true` if the token should be attached to the following token: i.e. no space between this token and the following token.
- `ATTACH_GROUP`: If this flag is `true` and if the `ATTACH_GROUP` flag for a previous and/or following token is `true`, then override the other spacing flags and put no space between the tokens in the group.

The `ATTACH_GROUP` is the least obvious flag. In English, a common use of this flag is for sequence of digits. If the user says four tokens "3" "point" "1" "4" (point='.'), and these tokens all have the `ATTACH_GROUP` flag set, then they are joined. The presentation string is "3.14". The name of the flag implies that tokens in the same "group" should have no spaces between them.

"previous" means the previously spoken token in the sequence of tokens - that is, previous in time. For left-to-right languages (e.g. English, German) `ATTACH_PREVIOUS` means left attachment - no space to the left. For right-to-left languages (e.g. Arabic, Hebrew) `ATTACH_PREVIOUS` means right attachment - no space to the right. The converse is true for `ATTACH_FOLLOWING`.

The spacing flags are OR'ed (Java '|' operator on integers). A legal value might be `(ATTACH_PREVIOUS | ATTACH_FOLLOWING)`. The `SEPARATE` value is 0 (zero). A flag can be tested by the following code:

```
// if attach previous ...
if ((token.getSpacingHint() & ResultToken.ATTACH_PREVIOUS) != 0)
  ...
```

`capitalizationHint` indicates how the written form of the *following* token should be capitalized.

The options are

- `CAP_AS_IS`: As-is indicates the capitalization of the spoken form of the following should not be changed
- `CAP_FIRST`: Capitalize first character of the spoken form of the following token
- `UPPERCASE`: All uppercase following token
- `LOWERCASE`: All lowercase following token

The Internationalized case conversion methods of the `java.lang.String` are recommended for implementing the capitalization hints.

Null Written Form

Some spoken directives to recognizers produce tokens that have no printable form. These tokens return null for the written form. Typically, these directives give explicit capitalization or spacing hints. The spoken forms of these tokens should be non-null (to allow the application to provide appropriate feedback to a user. Example directives for English might include:

- "Capitalize next", "Cap next", "Upper case"
- "Lowercase"
- "Uppercase"
- "No space"

For these tokens, the interpretation of the capitalization and spacing hints is specialized. If the spacing hint differs from the default, `SEPARATE`, it overrides the spacing hint of the next non-null token. If the capitalization hint differs from the default, `CAP_AS_IS`, it overrides the capitalization hints of previous non-null token (which in fact applies to the following token also).

Example

This example shows how a string of result tokens should be processed to produce a single printable string. The following is a sequence of tokens in a `FinalDictationResult` shown as spoken form, written form, and spacing and capitalization hints.

1. "NEW_LINE", "\u000A", SEPARATE, CAP_FIRST
2. "the", "the", SEPARATE, CAP_AS_IS
3. "UPPERCASE_NEXT", "", SEPARATE, UPPERCASE
4. "index", "index", SEPARATE, CAP_AS_IS
5. "is", "is" SEPARATE, CAP_AS_IS
6. "seven", "7", ATTACH_GROUP, CAP_AS_IS
7. "-", "-", ATTACH_GROUP, CAP_AS_IS
8. "two", "2", ATTACH_GROUP, CAP_AS_IS
9. "period", ".", ATTACH_PREVIOUS, CAP_FIRST

that could be converted to `"\nThe INDEX is 7-2."`

See Also:
Result, FinalResult, FinalDictationResult, FinalRuleResult

Field Summary

static int	**ATTACH_FOLLOWING** A `SpacingHint` flag set true when a token should be attached to the following token.
static int	**ATTACH_GROUP** A `SpacingHint` flag set true when a token should be attached to preceding and/or following tokens which also have the `ATTACH_GROUP` flag set true.
static int	**ATTACH_PREVIOUS** A `SpacingHint` flag set true when a token should be attached to the preceding token.
static int	**CAP_AS_IS** A `CapitalizationHint` indicating that the following word should be presented without changes in capitalization.
static int	**CAP_FIRST** A `CapitalizationHint` indicating that the following word should be presented with the first character in uppercase.
static int	**LOWERCASE** A `CapitalizationHint` indicating that the following word should be presented in lowercase.
static java.lang.String	**NEW_LINE** Special token representing the written form of the "New Line" directive.
static java.lang.String	**NEW_PARAGRAPH** Special token representing the written form of the "New Paragraph" directive.
static int	**SEPARATE** A `SpacingHint` returned when a token should be presented separately from surrounding tokens (preceding and following spaces).
static int	**UPPERCASE** A `CapitalizationHint` indicating that the following word should be presented in uppercase.

Method Summary

int	**getCapitalizationHint**() Get the capitalization hint.
long	**getEndTime**() Get the approximate end time for the token.
Result	**getResult**() Return a reference to the result that contains this token.
int	**getSpacingHint**() Get the spacing hints.
java.lang.String	**getSpokenText**() Get the spoken text of a token.
long	**getStartTime**() Get the approximate start time for the token.
java.lang.String	**getWrittenText**() Get the written form of a spoken token.

Field Detail

NEW_PARAGRAPH

`public static final java.lang.String NEW_PARAGRAPH`

Special token representing the written form of the "New Paragraph" directive. Equal to "\u2029". The spoken form of a "New Paragraph" directive may vary between recognizers.

NEW_LINE

`public static final java.lang.String NEW_LINE`

Special token representing the written form of the "New Line" directive. Equal to "\\n". The spoken form of a "New Line" directive may vary between recognizers.

SEPARATE

`public static final int SEPARATE`

A `SpacingHint` returned when a token should be presented separately from surrounding tokens (preceding and following spaces). Returned when `ATTACH_PREVIOUS`, `ATTACH_FOLLOWING`, and `ATTACH_GROUP` are all false. (See the description above.)

`SEPARATE` is the default spacing hint value.

Example:

```
if (resultToken.getSpacingHint() == ResultToken.SEPARATE)
  ...;
```

See Also:
getSpacingHint

ATTACH_PREVIOUS

`public static final int` **`ATTACH_PREVIOUS`**

A `SpacingHint` flag set true when a token should be attached to the preceding token. (See the description above.)

Example:

```
if ((resultToken.getSpacingHint() & ResultToken.ATTACH_PREVIOUS) != 0)
  ...;
```

See Also:
getSpacingHint

ATTACH_FOLLOWING

`public static final int` **`ATTACH_FOLLOWING`**

A `SpacingHint` flag set true when a token should be attached to the following token. (See the description above.)

Example:

```
if ((resultToken.getSpacingHint() & ResultToken.ATTACH_FOLLOWING) != 0)
  ...;
```

See Also:
getSpacingHint

ATTACH_GROUP

`public static final int` **`ATTACH_GROUP`**

A `SpacingHint` flag set true when a token should be attached to preceding and/or following tokens which also have the `ATTACH_GROUP` flag set true. (See the description above.)

Example:

```
if (((thisToken.getSpacingHint() & ResultToken.ATTACH_GROUP) != 0)
    && ((prevToken.getSpacingHint() & ResultToken.ATTACH_GROUP) != 0))
  ...;
```

See Also:
getSpacingHint

CAP_AS_IS

`public static final int` **`CAP_AS_IS`**

A `CapitalizationHint` indicating that the following word should be presented without changes in capitalization. This is the default value. (See the description above.)

CAP_FIRST

`public static final int` **`CAP_FIRST`**

A `CapitalizationHint` indicating that the following word should be presented with the first character in uppercase. (See the description above.)

UPPERCASE

`public static final int` **`UPPERCASE`**

A `CapitalizationHint` indicating that the following word should be presented in uppercase. (See the description above.)

LOWERCASE

`public static final int` **`LOWERCASE`**

A `CapitalizationHint` indicating that the following word should be presented in lowercase. (See the description above.)

Method Detail

getResult

`public Result` **`getResult`**`()`

Return a reference to the result that contains this token.

---

getSpokenText

`public java.lang.String` **`getSpokenText`**`()`

Get the spoken text of a token. In dictation, the spoken form is typically used when displaying unfinalized tokens. The difference between spoken and written forms is discussed above.

---

getWrittenText

`public java.lang.String getWrittenText()`

Get the written form of a spoken token. Spoken and written forms are discussed above.

---

getCapitalizationHint

`public int getCapitalizationHint()`

Get the capitalization hint. (See description above.) Values are `CAP_AS_IS` (the default), `CAP_FIRST`, `UPPERCASE`, `LOWERCASE`. Tokens from a `RuleGrammar` result always return `CAP_AS_IS`.

---

getSpacingHint

`public int getSpacingHint()`

Get the spacing hints. (See description above.) The value equals `SEPARATE` (the default) if the token should be presented with surrounding spaces. Otherwise any or all of the following flags can be true: `ATTACH_FOLLOWING`, `ATTACH_PREVIOUS`, `ATTACH_GROUP`. Tokens from a `RuleGrammar` result always return `SEPARATE`.

---

getStartTime

`public long getStartTime()`

Get the approximate start time for the token. The value is matched to the `System.currentTimeMillis()` time.

The start time of a token is always greater than or equal to the the end time of a preceding token. The values will be different if the tokens are separated by a pause.

Returns `-1` if timing information is not available. Not all recognizers provide timing information. Timing information is not usually available for unfinalized or finalized tokens in a `Result` that is not yet finalized. Even if timing information is available for the best-guess tokens, it might not be available for alternative tokens.

See Also:
currentTimeMillis

---

getEndTime

`public long getEndTime()`

Get the approximate end time for the token. The value is matched to the `System.currentTimeMillis()` time.

The end time of a token is always less than or equal to the the end time of a preceding token. The values will be different if the tokens are separated by a pause.

Returns `-1` if timing information is not available. Not all recognizers provide timing information. Timing information is not usually available for unfinalized or finalized tokens in a `Result` that is not yet finalized. Even if timing information is available for the best-guess tokens, it might not be available for alternative tokens.

See Also:
currentTimeMillis

Class javax.speech.recognition.Rule

```
java.lang.Object
  |
  +--javax.speech.recognition.Rule
```

Subclasses:
RuleAlternatives, RuleCount, RuleName, RuleParse, RuleSequence, RuleTag, RuleToken

---

public abstract class **Rule**
extends java.lang.Object
implements java.io.Serializable

A `Rule` object is the basic component of a `RuleGrammar` and represents anything that may appear on the right-hand side of a rule definition in Java Speech Grammar Format. Technically a `Rule` represents a JSGF "expansion".

`Rule` is an abstract class that is sub-classed by:

- `RuleAlternatives`: set of alternatives `Rule` objects
- `RuleCount`: contains a `Rule` that may occur optionally, zero or more times, or one or more times.
- `RuleName`: reference to a `Rule`
- `RuleSequence`: set of rules that occur in sequence
- `RuleTag`: contains a `Rule` tagged by a `String`
- `RuleToken`: reference to a token that may be spoken.

Another sub-class of `Rule` is `RuleParse` which is returned by the parse method of `RuleGrammar` to represent the structure of parsed text.

Any `Rule` object can be converted to a partial Java Speech Grammar Format String using its `toString` method.

See Also:
RuleAlternatives, RuleCount, RuleGrammar, RuleName, RuleParse, RuleSequence, RuleTag, RuleToken, Serialized Form

---

Constructor Summary

Rule()	

Method Summary

Rule	**copy**() Return a deep copy of a `Rule`.
java.lang.String	**toString**() Return a string representing the `Rule` in partial Java Speech Grammar Format.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Constructor Detail

Rule

```
public Rule()
```

Method Detail

copy

```
public abstract Rule copy()
```

Return a deep copy of a `Rule`. A deep copy implies that for a rule that contains other rules (i.e. `RuleAlternatives`, `RuleCount`, `RuleParse`, `RuleSequence`, `RuleTag`) the sub-rules are also copied. Note: copy differs from the typical use of `clone` because a clone is not normally a deep copy.

---

toString

```
public abstract java.lang.String toString()
```

Return a string representing the `Rule` in partial Java Speech Grammar Format. The `String` represents a portion of Java Speech Grammar Format that could appear on the right hand side of a rule definition.

Returns:
printable Java Speech Grammar Format string
Overrides:
toString in class java.lang.Object

Class javax.speech.recognition.RuleAlternatives

```
java.lang.Object
  |
  +--javax.speech.recognition.Rule
        |
        +--javax.speech.recognition.RuleAlternatives
```

---

public class **RuleAlternatives**
extends Rule

`RuleAlternatives` represents a `Rule` composed of a set of alternative sub-rules. `RuleAlternatives` are used to construct `RuleGrammar` objects. A `RuleAlternatives` object is spoken by saying one, and only one, of of its sub-rules.

A `RuleAlternatives` object contains a set of zero or more `Rule` objects. A set of zero alternatives is equivalent to `<VOID>` (it is unspeakable).

Weights may be (optionally) assigned to each alternative rule. The weights indicate the chance of each `Rule` being spoken. The `setWeights` method defines the constraints upon weights. If no weights are defined, then all alternatives are considered equally likely.

See Also:
Rule, RuleCount, RuleGrammar, RuleName, RuleParse, RuleSequence, RuleTag, RuleToken, VOID, Serialized Form

---

Field Summary

Rule[]	**rules** Set of alternative `Rule` objects.
float[]	**weights** Array of weights for each alternative `Rule` or `null` if the rules are equally likely.

Constructor Summary

RuleAlternatives(Rule[] rules, float[] weights) Construct a `RuleAlternatives` object with an array of sub-rules and an array of weights.
RuleAlternatives(Rule[] rules) Construct a `RuleAlternatives` object with an array of sub-rules.
RuleAlternatives(Rule rule) Construct a `RuleAlternatives` object containing a single sub-rule.
RuleAlternatives() Empty constructor creates zero-length list of alternatives.
RuleAlternatives(java.lang.String[] tokens) Constructor for `RuleAlternatives` that produces a phrase list from an array of `String` objects.

Method Summary

void	**append**(Rule rule) Append a single rule to the set of alternatives.
Rule	**copy**() Return a deep copy of this rule.
Rule[]	**getRules**() Return the array of alternative sub-rules.
float[]	**getWeights**() Return the array of weights.
void	**setRules**(Rule[] rules) Set the array of alternative sub-rules.
void	**setWeights**(float[] weights) Set the array of weights for the rules.
java.lang.String	**toString**() Return a `String` representing this object as partial Java Speech Grammar Format.

Methods inherited from class javax.speech.recognition.Rule
copy, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

rules

```
protected Rule[] rules
```

Set of alternative `Rule` objects.

---

weights

```
protected float[] weights
```

Array of weights for each alternative `Rule` or `null` if the rules are equally likely. If non-null, the `weights` array must have an identical length to the `rules` array.

Constructor Detail

RuleAlternatives

```
public RuleAlternatives(Rule[] rules,
                        float[] weights)
                 throws java.lang.IllegalArgumentException
```

Construct a `RuleAlternatives` object with an array of sub-rules and an array of weights. The rules array and weights array may be `null`. If the weights array is non-null, it must have identical length to the rules array.

Parameters:
`rules` - the set of alternative sub-rules
`set` - of weights for each rule or `null`
Throws:
java.lang.IllegalArgumentException - Error in length of array, or the weight values (see `setWeights`).
See Also:
setWeights

RuleAlternatives

```
public RuleAlternatives(Rule[] rules)
```

Construct a `RuleAlternatives` object with an array of sub-rules. The weights are set to `null`.

Parameters:
`rules` - the set of alternative sub-rules

RuleAlternatives

```
public RuleAlternatives(Rule rule)
```

Construct a `RuleAlternatives` object containing a single sub-rule. The weights array is set to `null`.

RuleAlternatives

```
public RuleAlternatives()
```

Empty constructor creates zero-length list of alternatives. Use the `setRules` method or append method to add alternatives.

A zero-length set of alternatives is equivalent to `<VOID>` (i.e. unspeakable).

RuleAlternatives

```
public RuleAlternatives(java.lang.String[] tokens)
```

Constructor for `RuleAlternatives` that produces a phrase list from an array of `String` objects. Each string is used to create a single `RuleToken` object.

A string containing multiple words (e.g. "san francisco") is treated as a single token. If appropriate, an application should parse such strings to produce separate tokens.

The phrase list may be zero-length or null. This will produce an empty set of alternatives which is equivalent to `<VOID>` (i.e. unspeakable).

Parameters:
`tokens` - a set of alternative tokens
See Also:
VOID

Method Detail

getRules

```
public Rule[] getRules()
```

Return the array of alternative sub-rules.

---

setRules

```
public void setRules(Rule[] rules)
```

Set the array of alternative sub-rules.

If the weights are non-null and the number of rules is not equal to the number of weights, the weights are set to `null`. To change the number of rules and weights, call `setRules` before `setWeights`.

See Also:

setWeights

append

`public void append(Rule rule)`

Append a single rule to the set of alternatives. The weights are set to `null`.

getWeights

`public float[] getWeights()`

Return the array of weights. May return `null`. If non-null, the length of the weights array is guaranteed to be the same length as the array of rules.

setWeights

```
public void setWeights(float[] weights)
              throws java.lang.IllegalArgumentException
```

Set the array of weights for the rules. The `weights` array may be `null`. If the weights are `null`, then all alternatives are assumed to be equally likely.

The length of the weights array must be the same length as the array of rules. The weights must all be greater than or equal to 0.0 and at least one must be non-zero.

To change the number of rules and weights, first call `setRules`.

Throws:

java.lang.IllegalArgumentException - Error in length of array or value of weights

See Also:

setRules

copy

`public Rule copy()`

Return a deep copy of this rule. See the `Rule.copy` documentation for an explanation of deep copy.

Overrides:

copy in class Rule

toString

`public java.lang.String toString()`

Return a `String` representing this object as partial Java Speech Grammar Format. The string is a legal right hand side of a rule definition.

Overrides:

toString in class Rule

Class javax.speech.recognition.RuleCount

```
java.lang.Object
  |
  +--javax.speech.recognition.Rule
        |
        +--javax.speech.recognition.RuleCount
```

public class **RuleCount**
extends Rule

Attaches a count to a contained `Rule` object to indicate the number of times it may occur. The contained rule may occur optionally (zero or one times), one or more times, or zero or more times. The three count are equivalent to the "[]", "+" and "*" operators of the Java Speech Grammar Format.

Any `Rule` not contained by a `RuleCount` object occurs once only.

See Also:
Rule, RuleAlternatives, RuleGrammar, RuleName, RuleParse, RuleSequence, RuleTag, RuleToken, Serialized Form

Field Summary

int	**count** Identifier for the rule count.
static int	**ONCE_OR_MORE** `ONCE_OR_MORE` indicates that the `Rule` may be spoken one or more times.
static int	**OPTIONAL** `OPTIONAL` indicates that the `Rule` is optional: zero or one occurrences.
Rule	**rule** The rule to which the count applies.
static int	**ZERO_OR_MORE** `ZERO_OR_MORE` indicates that the `Rule` may be spoken zero or more times.

Constructor Summary

RuleCount(Rule rule, int count) `RuleCount` constructor with contained rule and count.
RuleCount() Empty constructor sets rule to `null` and count to `OPTIONAL`.

Method Summary

Rule	**copy**() Return a deep copy of this rule.
int	**getCount**() Returns the count: `OPTIONAL`, `ZERO_OR_MORE`, `ONCE_OR_MORE`.
Rule	**getRule**() Returns the contained `Rule` object.
void	**setCount**(int count) Set the count.
void	**setRule**(Rule rule) Set the contained `Rule` object.
java.lang.String	**toString**() Return a string representing the `RuleCount` object in partial Java Speech Grammar Format.

Methods inherited from class javax.speech.recognition.Rule
copy, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

rule

`protected Rule rule`

The rule to which the count applies.

count

`protected int count`

Identifier for the rule count.

See Also:
OPTIONAL, ONCE_OR_MORE, ZERO_OR_MORE

OPTIONAL

`public static int OPTIONAL`

`OPTIONAL` indicates that the `Rule` is optional: zero or one occurrences. An optional `Rule` is surrounded by "[]" in Java Speech Grammar Format.

See Also:
getCount

ONCE_OR_MORE

`public static int ONCE_OR_MORE`

`ONCE_OR_MORE` indicates that the `Rule` may be spoken one or more times. This is indicated by the "+" operator in Java Speech Grammar Format.

See Also:
getCount

ZERO_OR_MORE

`public static int ZERO_OR_MORE`

`ZERO_OR_MORE` indicates that the `Rule` may be spoken zero or more times. This is indicated by the "*" operator in Java Speech Grammar Format.

See Also:
getCount

Constructor Detail

RuleCount

```
public RuleCount(Rule rule,
                 int count)
```

`RuleCount` constructor with contained rule and count.

RuleCount

`public RuleCount()`

Empty constructor sets rule to `null` and count to `OPTIONAL`.

Method Detail

getRule

`public Rule getRule()`

Returns the contained `Rule` object.

setRule

`public void setRule(Rule rule)`

Set the contained `Rule` object.

getCount

`public int getCount()`

Returns the count: `OPTIONAL`, `ZERO_OR_MORE`, `ONCE_OR_MORE`.

See Also:
OPTIONAL, ZERO_OR_MORE, ONCE_OR_MORE

setCount

`public void setCount(int count)`

Set the count. If count is not one of the defined values `OPTIONAL`, `ZERO_OR_MORE`, `ONCE_OR_MORE`) the call is ignored.

See Also:
OPTIONAL, ZERO_OR_MORE, ONCE_OR_MORE

copy

`public Rule copy()`

Return a deep copy of this rule. See the `Rule.copy` documentation for an explanation of deep copy.

Overrides:
copy in class Rule

toString

`public java.lang.String toString()`

Return a string representing the `RuleCount` object in partial Java Speech Grammar Format. The String represents the portion of Java Speech Grammar Format that could appear on the right hand side of a rule definition. Parenthesis will be placed around the contained `Rule` object if required. The output appears as one of:

```
[ruleString]   // OPTIONAL
ruleString *   // ZERO_OR_MORE
ruleString +   // ONCE_OR_MORE
```

Overrides:

toString in class Rule

Interface javax.speech.recognition.RuleGrammar

public abstract interface **RuleGrammar**
extends Grammar

`RuleGrammar` interface describes a `Grammar` that defines what users may say by a set rules. The rules may be defined as `Rule` objects that represent the rule in a data structure or defined in the Java Speech Grammar Format (JSGF).

All `RuleGrammars` are created and managed through the `Recognizer` interface. A `RuleGrammar` may be created with the `newRuleGrammar` method. A `RuleGrammar` is also created when JSGF text is loaded with the `loadJSGF` methods either from a `Reader` or from a `URL`.

A `RuleGrammar` contains the same information as a grammar definition in the Java Speech Grammar Format. That information includes:

- The name of the grammar (inherited from `Grammar` interface),
- A set of imports: each import references a single public rule of another `RuleGrammar` or all public rules of another `RuleGrammar`.
- A set of defined rules: each definition is identified by a unique rulename (unique within the `RuleGrammar`), a boolean flag indicating whether the rule is public, and a `Rule` object that provides the logical expansion of the rule (how it spoken).

The set of imports and the rule definitions can be changed by applications. For any change to take effect the application must call the `commitChanges` method of the `Recognizer`.

A `RuleGrammar` can be printed in Java Speech Grammar Format using the `toString` method. Individual `Rule` objects can be converted to JSGF with their `toString` methods.

In JSGF a rulename is surrounded by angle brackets (e.g. `<rulename>`). The angle brackets are ignored in calls to the methods of a `RuleGrammar` - they may be included but are stripped automatically and are not included in rulenames returned by `RuleGrammar` methods.

The rules defined in a `RuleGrammar` are either `public` or `private`. A `public` rule may be:

1. Imported into other `RuleGrammars`,
2. Enabled for recognition,
3. Or both.

When a `RuleGrammar` is enabled and when the activation conditions are appropriate (as described in the documentation for `Grammar`) then the `Grammar` is activated and any of the public rules of the grammar may be spoken. The `RuleGrammar` interface extends the enable methods of the `Grammar` interface to allow individual rules to be enabled and disabled independently.

A public rule may reference private rules and imported rules. Only the top public rule needs to be enabled for it to be spoken. The referenced rules (local private or imported public rules) do not need to be enabled to be spoken as part of the enabled rule.

See Also:
Recognizer, Rule, Grammar, Java Speech Grammar Format

Method Summary

void	**addImport**(RuleName importName) Import all rules or a specified rule from another grammar.
void	**deleteRule**(java.lang.String ruleName) Delete a rule from the grammar.
Rule	**getRuleInternal**(java.lang.String ruleName) Returns a reference to a recognizer’s internal rule object identified by a rule name.
Rule	**getRule**(java.lang.String ruleName) Returns a `Rule` object for a specified rulename.
boolean	**isEnabled**() Returns true if any public rule of a `RuleGrammar` is enabled.
boolean	**isEnabled**(java.lang.String ruleName) Test whether recognition of a specified rule of this `RuleGrammar` is enabled.
boolean	**isRulePublic**(java.lang.String ruleName) Test whether a rule is public.
RuleName[]	**listImports**() Return a list of the current imports.
java.lang.String[]	**listRuleNames**() List the names of all rules defined in this `RuleGrammar`.
RuleParse	**parse**(java.lang.String text, java.lang.String ruleName) Parse a `String` against a `RuleGrammar`.
RuleParse	**parse**(java.lang.String[] tokens, java.lang.String ruleName) Parse a sequence of tokens against a `RuleGrammar`.
RuleParse	**parse**(FinalRuleResult finalRuleResult, int nBest, java.lang.String ruleName) Parse the nth best result of a `FinalRuleResult` against a `RuleGrammar`.
void	**removeImport**(RuleName importName) Remove an import.
RuleName	**resolve**(RuleName ruleName) Resolve a rulename reference within a `RuleGrammar` to a fully-qualified rulename.
Rule	**ruleForJSGF**(java.lang.String JSGFText) Convert a `String` in partial Java Speech Grammar Format (JSGF) to a `Rule` object.
void	**setEnabled**(boolean enabled) Set the `enabled` property for a `RuleGrammar`.
void	**setEnabled**(java.lang.String ruleName, boolean enabled) Set the enabled property for a specified public rule.
void	**setEnabled**(java.lang.String[] ruleNames, boolean enabled) Set the enabled property for a set of public rules of a `RuleGrammar`.
void	**setRule**(java.lang.String ruleName, Rule rule, boolean isPublic) Set a rule in the grammar either by creating a new rule or updating an existing rule.
java.lang.String	**toString**() Return a string containing a specification for this `RuleGrammar` in Java Speech Grammar Format.

Method Detail

ruleForJSGF

```
public Rule ruleForJSGF(java.lang.String JSGFText)
                                       throws GrammarException
```

Convert a `String` in partial Java Speech Grammar Format (JSGF) to a `Rule` object. The string can be any legal *expansion* from JSGF: i.e. anything that can appear on the right hand side of a JSGF *rule definition*. For example,

```
Rule r = ruleGrammar.ruleForJSGF("[please] <action> all files");
```

Throws:
GrammarException - if the JSGF text contains any errors
See Also:
Java Speech Grammar Format

---

setRule

```
public void setRule(java.lang.String ruleName,
                    Rule rule,
                    boolean isPublic)
             throws java.lang.NullPointerException,
                    java.lang.IllegalArgumentException
```

Set a rule in the grammar either by creating a new rule or updating an existing rule. The rule being set is identified by its `"ruleName"` and defined by the `Rule` object and its `isPublic` flag. The `setRule` method is equivalent to a *rule definition* in the Java Speech Grammar Format (JSGF).

The change in the `RuleGrammar` takes effect when grammar changes are committed.

The rule object represents the *expansion* of a JSGF definition (the right hand side). It may be a `RuleToken`, `RuleName`, `RuleAlternatives`, `RuleSequence`, `RuleCount` or `RuleTag`. Each of these 6 object types is an extension of the `Rule` object. (The rule object cannot be an instance of `RuleParse` which is also an extension of `Rule`.)

A rule is most easily created from Java Speech Grammar Format text using the `ruleForJSGF` method. e.g.

```
gram.setRule(ruleName, gram.ruleForJSGF("open the <object>"), true);
```

The `isPublic` flag defines whether this rule may be enabled and active for recognition and/or imported into other rule grammars. It is equivalent to the `public` declaration in JSGF.

If the `Rule` object contains a fully-qualified reference to a rule of another `RuleGrammar`, an import is automatically generated for that rule if it is not already imported. A subsequent call to `listImports` will return that import statement.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then they are automatically stripped.

The `Rule` object passed to the `RuleGrammar` is copied before being applied to recognizer. Thus, an application can modify and re-use a `rule` object without unexpected side-effects. Also, a different object will be returned by the `getRule` and `getRuleInternal` methods (although it will contain the same information).

Parameters:
`ruleName` - unique name of rule being defined (unique for this `RuleGrammar`)
`rule` - logical expansion for the rulename
`isPublic` - true if this rule can be imported into other `RuleGrammars` or enabled
Throws:
java.lang.IllegalArgumentException - if rule is not a legal instance of Rule
java.lang.NullPointerException - if ruleName or rule are null
See Also:
ruleForJSGF, getRule, getRuleInternal, commitChanges, RuleAlternatives, RuleCount, RuleName, RuleSequence, RuleTag, RuleToken

---

getRule

```
public Rule getRule(java.lang.String ruleName)
```

Returns a `Rule` object for a specified rulename. Returns `null` if the rule is unknown. The returned object is a copy of the recognizer's internal object so it can be modified in any way without affecting the recognizer. The `setRule` method should be called when a change to the returned rule object needs to be applied to the recognizer.

The `Rule.toString` method can be used to convert the return object to a printable String in Java Speech Grammar Format.

If there is a rule structure currently pending for a `commitChanges` that structure is returned. Otherwise, the current structure being used by the recognizer on incoming speech is returned.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

If fast, read-only access to the rule object is required (e.g. in parsing), then the application may use `getRuleInternal`.

Parameters:
`ruleName` - the `rulename` to be returned
Returns:
the definition of `ruleName` or `null`
See Also:
setRule, getRuleInternal, commitChanges

---

getRuleInternal

```
public Rule getRuleInternal(java.lang.String ruleName)
```

Returns a reference to a recognizer's internal rule object identified by a rule name. The application should *never* modify the returned object. This method is intented for use by parsers and other software that needs to quickly analyse a recognizer's grammars without modifying them (without the overhead of making copies, as required by `getRule`). If the returned object is ever modified in any way, `getRule` and `setRule` should be used.

Returns `null` if the rule is unknown.

If there is a `Rule` structure currently pending for a `commitChanges` that structure is returned. Otherwise, the current structure being used by the recognizer on incoming speech is returned.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

See Also:
setRule, getRule, commitChanges

---

isRulePublic

```
public boolean isRulePublic(java.lang.String ruleName)
                    throws java.lang.IllegalArgumentException
```

Test whether a rule is public. Public rules may be enabled to be activated for recognition and/or may be imported into other `RuleGrammars`.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

Parameters:
`ruleName` - the rulename being tested
Returns:
true if `ruleName` is public
Throws:
java.lang.IllegalArgumentException - if ruleName is unknown

---

listRuleNames

```
public java.lang.String[] listRuleNames()
```

List the names of all rules defined in this `RuleGrammar`. If any rules are pending deletion they are not listed (between a call to `deleteRule` and a `commitChanges` taking effect).

The returned names do not include the <> symbols.

---

deleteRule

```
public void deleteRule(java.lang.String ruleName)
               throws java.lang.IllegalArgumentException
```

Delete a rule from the grammar. The deletion only takes effect when grammar changes are next committed.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

Parameters:
`ruleName` - name of the defined rule to be deleted
Throws:
java.lang.IllegalArgumentException - if ruleName is unknown
See Also:
commitChanges

---

setEnabled

```
public void setEnabled(boolean enabled)
```

Set the `enabled` property for a `RuleGrammar`. When a grammar is enabled and when the activation conditions are appropriate, the `RuleGrammar` is activated for recognition and users may speak the commands/words/phrases defined by the grammar and results will be generated. The enabled state of a grammar can be tested by the `isEnabled` method. The activation state of a grammar can be tested by the `isActive` methods.

Recognizers can have multiple `Grammars` enabled and active at any time. A change in enabled status of the `Grammar` only takes effect when grammar changes are next committed. and the `CHANGES_COMMITTED` event is issued. The grammar is only activated once all the activation conditions are met (see documentation for `Grammar`).

All enabled rules of a `RuleGrammar` share the same `ActivationMode`. Thus, when a `RuleGrammar` is activated or deactivated for recognition, all the enabled rules of the `RuleGrammar` are activated or deactivated together.

This method is inherited from the `Grammar` interface. For a `RuleGrammar`, `setEnabled(true)` enables all *public* rules of the grammar. `setEnabled(false)` disables all public rules of the `RuleGrammar`. A `RuleGrammar` also provides `setEnabled` methods for enabling and disabling individual public rules or sets of public rules.

It is not an error to enable an enabled grammar or disable an disable grammar.

Specified by:
setEnabled in interface Grammar
See Also:
setEnabled(String, boolean), setEnabled(String[], boolean), isEnabled, setActivationMode

---

setEnabled

```
public void setEnabled(java.lang.String ruleName,
                       boolean enabled)
               throws java.lang.IllegalArgumentException
```

Set the enabled property for a specified public rule. This method behaves the same as the setEnabled(boolean) method except that it affects only a single public rule. The enabled state of other rules is unchanged.

Individual rules of a `RuleGrammar` may be individually enabled and disabled. Once any rule is enabled, the `RuleGrammar` is considered to be enabled. If all rules are disabled, then the `RuleGrammar` is considered disabled.

A change in the enabled property of a `Rule` or a `RuleGrammar` only takes effect when grammar changes are next committed.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

Parameters:
`ruleName` - name of the rule to be enabled or disabled
`enabled` - `true` to enable `ruleName`, `false` to disable
Throws:
java.lang.IllegalArgumentException - if `ruleName` is unknown or if it is a non-public rule
See Also:
setEnabled(boolean), setEnabled(String[], boolean), commitChanges

---

setEnabled

```
public void setEnabled(java.lang.String[] ruleNames,
                       boolean enabled)
               throws java.lang.IllegalArgumentException
```

Set the enabled property for a set of public rules of a `RuleGrammar`. This method behaves the same as the setEnabled(boolean) method except that it only affects a defined single public rule. This call does not affect the enabled state of other public rules of the `RuleGrammar`.

If any one or more rules are enabled, the `RuleGrammar` is considered to be enabled. If all rules are disabled, then the `RuleGrammar` is considered disabled.

A change in the enabled property of `Rules` or a `RuleGrammar` only takes effect when grammar changes are next committed.

If any rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

Parameters:
`ruleNames` - the set of rulenames to be enabled or disabled
`enabled` - `true` to enabled rulenames, `false` to disable
Throws:
java.lang.IllegalArgumentException - if one or more `ruleNames` is unknown or if any is a non-public rule
See Also:
commitChanges

---

isEnabled

```
public boolean isEnabled()
```

Returns true if any public rule of a `RuleGrammar` is enabled.

Specified by:
isEnabled in interface Grammar
Returns:
`true` if `RuleGrammar` is enabled, otherwise `false`

---

isEnabled

```
public boolean isEnabled(java.lang.String ruleName)
                 throws java.lang.IllegalArgumentException
```

Test whether recognition of a specified rule of this `RuleGrammar` is enabled. If `ruleName` is null, the method is equivalent to `isEnabled()`.

If the rule name contains both start/end angle brackets (e.g. `"<ruleName>"`), then the brackets are ignored.

Parameters:
`ruleName` - name of the rule being tested
Returns:
`true` if `ruleName` is enabled, otherwise `false`
Throws:
java.lang.IllegalArgumentException - if ruleName is unknown or if it is a non-public rule

---

resolve

```
public RuleName resolve(RuleName ruleName)
                                        throws GrammarException
```

Resolve a rulename reference within a `RuleGrammar` to a fully-qualified rulename. The input rulename may be a simple rulename, qualified or fully-qualified. If the rulename cannot be resolved, the method returns `null`.

If the rulename being resolved is a local reference, the return value is a fully-qualified rulename with its grammar part being the name of this `RuleGrammar`.

Example: in a grammar that imports the rule <number> from the grammar `"com.sun.examples"`, the following would return the fully-qualified rulename `com.sun.examples.number`.

```
gram.resolve(new RuleName("number"));
```

If the input rulename is a fully-qualified rulename, then the method checks whether that rulename exists (and could therefore be successfully referenced in this `RuleGrammar`). If the rulename exists, then the retun value is the same as the input value, otherwise the method returns `null`.

Parameters:
`ruleName` - reference to rulename to be resolved
Returns:
fully-qualified reference to a rulename
Throws:
GrammarException - if an error is found in the definition of the `RuleGrammar` or if `rulename` is an ambiguous reference

---

addImport

```
public void addImport(RuleName importName)
```

Import all rules or a specified rule from another grammar. The `RuleName` should be in one these forms:

```
<package.grammar.ruleName>
<package.grammar.*>
<grammar.ruleName>
```

<grammar.*>

which are equivalent to the following declarations in the Java Speech Grammar Format.

```
// import all public rules of a grammar
import <package.grammar.*>
import <grammar.*>

// import a specific public rule name of a grammar
import <package.grammar.ruleName>
import <grammar.ruleName>
```

The forms without a package are only legal when the *full grammar name* does not include a package name.

The `addImport` takes effect when grammar changes are committed. When changes are committed, all imports must be resolvable. Specifically, every `RuleGrammar` listed in an import must exist, and every fully qualified rulename listed in an import must exist. If any ommissions are found, the `commitChanges` method throws an exception and the changes do not take effect.

It is not an exception to import a rule or set of rules and not reference them.

See Also:
listImports, removeImport, commitChanges

---

removeImport

```
public void removeImport(RuleName importName)
                 throws java.lang.IllegalArgumentException
```

Remove an import. The name follows the format of `addImport`.

The change in imports only takes effect when grammar changes are committed.

Throws:
java.lang.IllegalArgumentException - if importName is not currently imported
See Also:
addImport, commitChanges

---

listImports

```
public RuleName[] listImports()
```

Return a list of the current imports. Returns zero length array if no `RuleGrammars` are imported.

See Also:
addImport, removeImport

---

parse

```
public RuleParse parse(java.lang.String text,
                       java.lang.String ruleName)
                                 throws GrammarException
```

Parse a `String` against a `RuleGrammar`. Parsing is the process of matching the text against the rules that are defined in the grammar. The text is tokenized as a Java Speech Grammar Format string (white-spacing and quotes are significant).

The string is parsed against the rule identified by `ruleName`, which may identify any defined rule of this rule grammar (public or private, enabled or disabled). If the `ruleName` is `null`, the string is parsed against *all enabled* rules of the grammar.

The method returns a `RuleParse` object. The documentation `RuleParse` describes how the parse structure and grammar structure correspond.

If parse fails, then the return value is `null`. To succeed, the parse must match the complete input string.

For some grammars and strings, multiple parses are legal. The parse method returns only one. It is not defined which of the legal parses should be returned. Development tools will help to analyse grammars for such ambiguities. Also grammar libraries can be used to parse results to check for these ambiguities.

Throws:
GrammarException - if an error is found in the definition of the `RuleGrammar`
See Also:
parse(String[], String), parse(FinalRuleResult, int, String)

---

parse

```
public RuleParse parse(java.lang.String[] tokens,
                       java.lang.String ruleName)
                                 throws GrammarException
```

Parse a sequence of tokens against a `RuleGrammar`. In all other respects this parse method is equivalent to the `parse(String text, String ruleName)` method except that the string is pre-tokenized.

Throws:
GrammarException - if an error is found in the definition of the `RuleGrammar`
See Also:
parse(String, String), parse(FinalRuleResult, int, String)

---

parse

```
public RuleParse parse(FinalRuleResult finalRuleResult,
                       int nBest,
                       java.lang.String ruleName)
                                 throws GrammarException
```

Parse the nth best result of a `FinalRuleResult` against a `RuleGrammar`. In other respects this parse method is equivalent to the `parse(String text, String ruleName)` method described above.

A rejected result (`REJECTED` state) is not guaranteed to parse. Also, if the `RuleGrammar` has been modified since the result was issued, parsing is not guaranteed to succeed.

Throws:
GrammarException - if an error is found in the definition of the `RuleGrammar`
See Also:
parse(String, String), parse(String[], String)

---

toString

```
public java.lang.String toString()
```

Return a string containing a specification for this `RuleGrammar` in Java Speech Grammar Format. The string includes the grammar name declaration, import statements and all the rule definitions. When sending JSGF to a stream (e.g. a file) the application should prepend the header line with the appropriate character encoding information:

```
#JSGF V1.0 encoding-format;
```

Overrides:
toString in class java.lang.Object

See Also:
Java Speech Grammar Format

Class javax.speech.recognition.RuleName

```
java.lang.Object
  |
  +--javax.speech.recognition.Rule
        |
        +--javax.speech.recognition.RuleName
```

public class **RuleName**
extends Rule

A `RuleName` is a reference to a named rule. A `RuleName` is equivalent to the various forms of rulename syntax in the Java Speech Grammar Format (JSGF).

A *fully-qualified rulename* consists of a *full grammar name* plus the *simple rulename*. A full grammar name consists of a *package name* and a *simple grammar name*. The three legal forms of a rulename allowed by the Java Speech Grammar Format and in the `RuleName` object are:

- Simple rulename: `<simpleRulename>`
  e.g. `<digits>`, `<date>`
- Qualified rulename: `<simpleGrammarName.simpleRulename>`
  e.g. `<numbers.digits>`, `<places.cities>`
- Fully-qualified rulename: `<packageName.simpleGrammarName.simpleRulename>`
  e.g. `<com.sun.numbers.digits>` `<com.acme.places.zipCodes>`

The full grammar name is the following portion of the fully-qualified rulename: `packageName.simpleGrammarName`. For example, `<com.sun.numbers>` `<com.acme.places>`.

There are two special rules are defined in JSGF, `<NULL>` and `<VOID>`. Both have static instances in this class for convenience. These rulenames can be referenced in any grammar without an import statement.

There is a special case of using a `RuleName` to declare and manage imports of a `RuleGrammar`. This form is used with the `addImport` and `removeImport` methods of a `RuleGrammar`. It requires a full grammar name plus the string `"*"` for the simple rulename. For example: `<com.acme.places.*>`

The angle brackets placed around rulenames are syntactic constructs in JSGF but are not a part of the rulename. For clarity of code, the angle brackets may be included in calls to this class but they are automatically stripped.

The following referencing and name resolution conditions of JSGF apply.

- Any rule in a local grammar may be referenced with a simple rulename, qualified rulename or fully-qualified rulename.
- A public rule of another grammar may be referenced by its simple rulename or qualified rulename if the rule is imported and the name is not ambiguous with another imported rule.
- A public rule of another grammar may be referenced by its fully-qualified rulename with or without a corresponding import statement.

See Also:
Rule, addImport, Serialized Form

Field Summary

java.lang.String	**fullRuleName** The complete specified rulename.
static RuleName	**NULL** Special `<NULL>` rule of JSGF defining a rule that is always matched.
java.lang.String	**packageName** The rule’s package name or `null` if not specified.
java.lang.String	**simpleGrammarName** The rule’s simple grammar name or `null` if not specified.
java.lang.String	**simpleRuleName** The simple rulename.
static RuleName	**VOID** Special `<VOID>` rule of JSGF defining a rule that can never be matched.

Constructor Summary

RuleName(java.lang.String ruleName) Construct a `RuleName` from a string.
RuleName() Empty constructor which sets the rule to `<NULL>`.
RuleName(java.lang.String packageName, java.lang.String simpleGrammarName, java.lang.String simpleRuleName) Construct a `RuleName` from its package-, grammar- and simple-name components.

Method Summary

Rule	**copy**() Return a deep copy of this rule.
java.lang.String	**getFullGrammarName**() Get the full grammar name.
java.lang.String	**getPackageName**() Get the rule’s package name.
java.lang.String	**getRuleName**() Get therulename including the package and grammar name if they are non-null.
java.lang.String	**getSimpleGrammarName**() Get the simple grammar name.
java.lang.String	**getSimpleRuleName**() Get the simple rulename.
boolean	**isLegalRuleName**() Tests whether this `RuleName` is a legal JSGF rulename.
static boolean	**isLegalRuleName**(java.lang.String name) Tests whether a string is a legal JSGF rulename format.
static boolean	**isRuleNamePart**(char c) Tests whether a character is a legal part of a Java Speech Grammar Format rulename.
void	**setRuleName**(java.lang.String ruleName) Set the rulename.
void	**setRuleName**(java.lang.String packageName, java.lang.String simpleGrammarName, java.lang.String simpleRuleName) Set the rule’s name with package name, simple grammar name and simple rulename components.
java.lang.String	**toString**() Return a String representing the `RuleName` as partial Java Speech Grammar Format text.

Methods inherited from class javax.speech.recognition.Rule
copy, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notifyAll, notify, toString, wait, wait, wait

Field Detail

fullRuleName

`protected java.lang.String` **`fullRuleName`**

The complete specified rulename. Maybe a fully-qualified rulename, qualified rulename, or simple rulename depending upon how the object is constructed.

---

packageName

`protected java.lang.String` **`packageName`**

The rule's package name or `null` if not specified.

---

simpleGrammarName

`protected java.lang.String` **`simpleGrammarName`**

The rule's simple grammar name or `null` if not specified.

---

simpleRuleName

`protected java.lang.String` **`simpleRuleName`**

The simple rulename.

NULL

`public static RuleName` **`NULL`**

Special `<NULL>` rule of JSGF defining a rule that is always matched.

VOID

`public static RuleName` **`VOID`**

Special `<VOID>` rule of JSGF defining a rule that can never be matched.

Constructor Detail

RuleName

`public` **`RuleName`**`(java.lang.String ruleName)`

Construct a `RuleName` from a string. Leading and trailing angle brackets are stripped if found. The rulename may be a simple rulename, qualified rulename or full-qualified rulename.

RuleName

`public` **`RuleName`**`()`

Empty constructor which sets the rule to `<NULL>`.

RuleName

```
public RuleName(java.lang.String packageName,
                java.lang.String simpleGrammarName,
                java.lang.String simpleRuleName)
         throws java.lang.IllegalArgumentException
```

Construct a `RuleName` from its package-, grammar- and simple-name components. Leading and trailing angle brackets are stripped from `ruleName` if found. The package name may be `null`. The grammar name may be `null` only if `packageName` is `null`.

Parameters:
`packageName` - the package name of a fully-qualified rulename or `null`
`simpleGrammarName` - the grammar name of a fully-qualified or qualified rule or `null`
`simpleRuleName` - the simple rulename
Throws:
java.lang.IllegalArgumentException - null `simpleGrammarName` with non-null `packageName`

Method Detail

getRuleName

`public java.lang.String` **`getRuleName`**`()`

Get therulename including the package and grammar name if they are non-null. The return value may be a fully-qualified rulename, qualified rulename, or simple rulename.

---

setRuleName

`public void` **`setRuleName`**`(java.lang.String ruleName)`

Set the rulename. The rulename may be a simple-, qualified- or fully-qualified rulename. Leading and trailing angle brackets are stripped if found.

---

setRuleName

```
public void setRuleName(java.lang.String packageName,
                        java.lang.String simpleGrammarName,
                        java.lang.String simpleRuleName)
                 throws java.lang.IllegalArgumentException
```

Set the rule's name with package name, simple grammar name and simple rulename components. Leading and trailing angle brackets are stripped from `ruleName`. The package name may be `null`. The simple grammar name may be null only if `packageName` is `null`.

Throws:
java.lang.IllegalArgumentException - null `simpleGrammarName` with non-null `packageName`

---

getSimpleRuleName

```
public java.lang.String getSimpleRuleName()
```

Get the simple rulename.

---

getSimpleGrammarName

```
public java.lang.String getSimpleGrammarName()
```

Get the simple grammar name. May be `null`.

---

getFullGrammarName

```
public java.lang.String getFullGrammarName()
```

Get the full grammar name. If the `packageName` is `null`, the return value is the simple grammar name. May return `null`.

---

getPackageName

```
public java.lang.String getPackageName()
```

Get the rule's package name.

---

isLegalRuleName

```
public boolean isLegalRuleName()
```

Tests whether this `RuleName` is a legal JSGF rulename. The `isLegalRuleName(java.lang.String)` method defines legal rulename forms.

See Also:
isLegalRuleName(java.lang.String), isRuleNamePart(char)

---

isLegalRuleName

```
public static boolean isLegalRuleName(java.lang.String name)
```

Tests whether a string is a legal JSGF rulename format. The legal patterns for rulenames are defined above. The method does not test whether the rulename exists or is resolvable (see the `resolve` method of `RuleGrammar`).

An import string (e.g. "com.acme.*") is considered legal even though the "*" character is not a legal rulename character. If present, starting and ending angle brackets are ignored.

See Also:
isLegalRuleName(), isRuleNamePart(char)

---

isRuleNamePart

```
public static boolean isRuleNamePart(char c)
```

Tests whether a character is a legal part of a Java Speech Grammar Format rulename.

See Also:
isLegalRuleName(), isLegalRuleName(java.lang.String)

---

copy

```
public Rule copy()
```

Return a deep copy of this rule.
Overrides:
copy in class Rule

---

toString

```
public java.lang.String toString()
```

Return a String representing the `RuleName` as partial Java Speech Grammar Format text. The return value is `<RuleName>`.
Overrides:
toString in class Rule

Class javax.speech.recognition.RuleParse

```
java.lang.Object
   |
   +--javax.speech.recognition.Rule
           |
           +--javax.speech.recognition.RuleParse
```

---

public class **RuleParse**
extends Rule

Represents the output of a parse of a `Result` or a string against a `RuleGrammar`. The `RuleParse` object indicates how the result or text matches to the rules of the `RuleGrammar` and the rules imported by that grammar.

The `RuleParse` structure returned by parsing closely matches the structure of the grammar it is parsed against: if the grammar contains `RuleTag`, `RuleSequence`, `RuleToken` objects and so on, the returned `RuleParse` will contain paired objects.

The `RuleParse` object itself represents the match of text to a named rule or to any rule referenced within a rule. The rulename field of the `RuleParse` is the fully-qualified name of the rule being matched. The `Rule` field of the `RuleParse` represents the parse structure (how that rulename is matched).

The expansion (or logical structure) of a `RuleParse` matches the structure of the definition of the rule being parsed. The following indicates the mapping of an entity in the rule being parsed to the paired object in the `RuleParse`.

- `RuleAlternatives`: maps to a `RuleAlternatives` object containing a single `Rule` object for the one entity in the set of alternatives that is matched.
- `RuleCount`: maps to a `RuleSequence` containing a `Rule` for each match of the rule contained by `RuleCount`. The sequence may contain zero, one or multiple rule matches (for optional, zero-or-more or one-or-more operators).
- `RuleName`: maps to a `RuleParse` indicating how the referenced rule was matched. The rulename field of the `RuleParse` is the matched `RuleName`. The exception for `NULL` is described below.
- `RuleSequence`: maps to a `RuleSequence` with a matching rule for each rule in the original sequence.