---

package javax.media.protocol

Interface Index

- Controls
- Positionable
- PullSourceStream
- PushSourceStream
- RateConfiguration
- RateConfigureable
- Seekable
- SourceStream
- SourceTransferHandler

Class Index

- ContentDescriptor
- DataSource
- PullDataSource
- PushDataSource
- RateRange
- URLDataSource

Interface javax.media.CachingControl

public interface **CachingControl**
extends Control

`CachingControl` is an interface supported by `Players` that are capable of reporting download progress. Typically, this control is accessed through the `Controller.getControls` method. A `Controller` that supports this control will post `CachingControlEvents` often enough to support the implementation of custom progress GUIs.

Version:
1.18, 97/08/25.
See Also:
Controller, ControllerListener, CachingControlEvent, Player

Variable Index

o **LENGTH_UNKNOWN**
Use to indicate that the `CachingControl` doesn't know how long the content is.

The definition is: LENGTH_UNKNOWN == Long.MAX_VALUE

Method Index

o **getContentLength**()
Get the total number of bytes in the media being downloaded.
o **getContentProgress**()
Get the total number of bytes of media data that have been downloaded so far.
o **getControlComponent**()
Get a `Component` that provides additional download control.
o **getProgressBarComponent**()
Get a `Component` for displaying the download progress.
o **isDownloading**()
Check whether or not media is being downloaded.

Variables

o **LENGTH_UNKNOWN**

```
public static final long LENGTH_UNKNOWN
```

Use to indicate that the `CachingControl` doesn't know how long the content is.

The definition is: LENGTH_UNKNOWN == Long.MAX_VALUE

Methods

o **isDownloading**

```
public abstract boolean isDownloading()
```

Check whether or not media is being downloaded.

Returns:
Returns `true` if media is being downloaded; otherwise returns `false`. .

o **getContentLength**

```
public abstract long getContentLength()
```

Get the total number of bytes in the media being downloaded. Returns `LENGTH_UNKNOWN` if this information is not available.

Returns:
The media length in bytes, or `LENGTH_UNKNOWN`.

o **getContentProgress**

```
public abstract long getContentProgress()
```

Get the total number of bytes of media data that have been downloaded so far.

Returns:
The number of bytes downloaded.

o **getProgressBarComponent**

```
public abstract Component getProgressBarComponent()
```

Get a `Component` for displaying the download progress.

Returns:
Progress bar GUI.

o **getControlComponent**

```
public abstract Component getControlComponent()
```

Get a `Component` that provides additional download control. Returns `null` if only a progress bar is provided.

Returns:
Download control GUI.

Class javax.media.CachingControlEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.CachingControlEvent
```

public class **CachingControlEvent**
extends ControllerEvent

This event is generated by a `Controller` that supports the `CachingControl` interface. It is posted when the caching state changes.

Version:
1.10, 97/08/23.
See Also:
Controller, ControllerListener, CachingControl

Constructor Index

o **CachingControlEvent**(Controller, CachingControl, long)
Construct a `CachingControlEvent` from the required elements.

Method Index

o **getCachingControl**()
Get the `CachingControl` object that generated the event.
o **getContentProgress**()
Get the total number of bytes of media data that have been downloaded so far.

Constructors

o **CachingControlEvent**

```
public CachingControlEvent(Controller from,
                           CachingControl cacheControl,
                           long progress)
```

Construct a `CachingControlEvent` from the required elements.

Methods

o **getCachingControl**

```
public CachingControl getCachingControl()
```

Get the `CachingControl` object that generated the event.

Returns:
The `CachingControl` object.

o **getContentProgress**

```
public long getContentProgress()
```

Get the total number of bytes of media data that have been downloaded so far.

Returns:
The number of bytes of media data downloaded.

Interface javax.media.Clock

public interface **Clock**

The Clock interface is implemented by objects that support the Java Media time model. For example, this interface might be implemented by an object that decodes and renders MPEG movies.

Clock and TimeBase

A Clock contains a TimeBase that provides a source of time, much like a crystal oscillator. The only information that a TimeBase provides is its current time; it does not provide any methods for influencing how time is kept. A Clock defines a transformation on the time that its TimeBase keeps, typically marking time for a particular media stream. The time that a Clock keeps is referred to as the *media time*.

Clock Transform

The transformation that a Clock defines on a TimeBase is defined by three parameters: rate, *media start−time* (mst), and *time−base start−time* (tbst). Given a *time−base time* (tbt), the *media time* (mt) can be calculated using the following transformation:

```
mt = mst + (tbt − tbst)*rate
```

The rate is simply a scale factor that is applied to the TimeBase. For example, a rate of 2.0 indicates that the Clock will run at twice the rate of its TimeBase. Similarly, a negative rate indicates that the Clock runs in the opposite direction of its TimeBase.

The *time−base start−time* and the *media start−time* define a common point in time at which the Clock and the TimeBase are synchronized.

Default Time Base

A Clock has a default TimeBase. For many objects that support the Clock interface, the default TimeBase is the system TimeBase. The system TimeBase can be obtained from Manager through the getSystemTimeBase method.

Some Clocks have a TimeBase other than the system TimeBase. For example, an audio renderer that implements the Clock interface might have a TimeBase that represents a hardware clock.

Using a Clock

You can get the TimeBase associated with a Clock by calling the getTimeBase method. To change the TimeBase that a Clock uses, you call the setTimeBase method. These get and set methods can be used together to synchronize different Clocks to the same TimeBase.

For example, an application might want to force a video renderer to sync to the TimeBase of an audio renderer. To do this, the application would call getTimeBase on the audio renderer and then use the value returned to call setTimeBase on the video renderer. This would ensure that the two rendering objects use the same source of time. You can reset a Clock to use its default TimeBase by calling setTimeBase(null).

Some Clocks are incapable of using another TimeBase. If this is the case, an IncompatibleTimeBaseException is thrown when setTimeBase is called.

Clock also provides methods for getting and setting a Clock's *media time* and rate:

- getMediaTime and setMediaTime
- getRate and setRate

Starting a Clock

Until a Clock's TimeBase transformation takes effect, the Clock is in the *Stopped* state. Once all three transformation parameters (*media start−time*, *time−base start−time*, and rate) have been provided to the Clock, it enters the *Started* state.

To start a Clock, syncStart is called with the *time−base start−time* as an argument. The new *media start−time* is taken as the current *media time*, and the current rate defines the Clock's rate parameter. When syncStart is called, the Clock and its TimeBase are locked in sync and the Clock is considered to be in the *Started* state.

When a Clock is stopped and then restarted (using syncStart), the *media start−time* for the restarted Clock is the current *media time*. The syncStart method is often used to synchronize two Clocks that share the same TimeBase. When the *time−base start−time* and rate of each clock are set to the same values and each Clock is set with the appropriate *media start−time*, the two Clocks will run in sync.

When syncStart is called with a new *time−base start−time*, the synchronization with the *media time* doesn't occur until the TimeBase reaches the *time−base start−time*. The getMediaTime method returns the untransformed *media time* until the TimeBase reaches the *time−base start−time*.

The getSyncTime method behaves slightly differently. Once syncStart is invoked, getSyncTime always reports the transformed *time−base time*, whether or not the *time−base start−time* has been reached. You can use getSyncTime to determine how much time remains before the *time−base start−time* is reached. When the *time−base start−time* is reached, both getMediaTime and getSyncTime return the same value.

Objects that implement the Clock interface can provide more convenient start methods than syncStart. For example, Player defines start, which should be used instead of syncStart to start a Player.

Stopping a Clock

A *Stopped* `Clock` is no longer synchronized to its `TimeBase`. When a `Clock` is *Stopped*, its *media time* no longer moves in rate−adjusted synchronization with the *time−base time* provided by its `TimeBase`.

There are two ways to explicitly stop a `Clock`: you can invoke `stop` or set a *media stop−time*. When `stop` is invoked, synchronization with the `TimeBase` immediately stops. When a *media stop−time* is set, synchronization stops when the *media stop−time* passes.

A `Clock's` rate affects how its *media stop−time* is interpreted. If its rate is positive, the `Clock` stops when the *media time* becomes greater than or equal to the stop time. If its rate is negative, the `Clock` stops when the *media time* becomes less than or equal to the stop time.

If the stop−time is set to a value that the `Clock` has already passed, the `Clock` immediately stops.

Once a stop−time is set, it remains in effect until it is changed or cleared. To clear a stop−time, call `setStopTime` with `Clock.RESET`. A `Clock's` stop−time is cleared automatically when it stops.

If no stop−time is ever set or if the stop−time is cleared, the only way to stop the `Clock` is to call the `stop` method.

Clock State

Conceptually, a `Clock` is always in one of two states: *Started* or *Stopped*. A `Clock` enters the *Started* state after `syncStart` has been called and the `Clock` is mapped to its `TimeBase`. A `Clock` returns to the *Stopped* state immediately when the `stop` method is called or the *media time* passes the stop time.

Certain methods can only be invoked when the `Clock` is in a particular state. If the `Clock` is in the wrong state when one of these methods is called, an error or exception is thrown.

Methods Restricted to *Started* Clocks

The `mapToTimeBase` method can only be called on a `Clock` in the *Started* state. If it is invoked on a *Stopped* `Clock`, a `ClockStoppedException` is thrown. This is because the `Clock` is not synchronized to a `TimeBase` when it is *Stopped*.

Methods Restricted to *Stopped* Clocks

The following methods can only be called on a `Clock` in the *Stopped* state. If invoked on a *Started* `Clock`, these methods throw a `ClockStartedError`.

- `syncStart`
- `setTimeBase`
- `setMediaTime`
- `setRate`

Resetting the rate, the *media time*, the time base, or the *time−base start−time* implies a complete remapping between the `Clock` and its `TimeBase` and is not allowed on a *Started* `Clock`.

Methods with Additional Restrictions

A race condition occurs if a new *media stop−time* is set when a `Clock` is already approaching a previously set *media stop−time*. In this situation, it impossible to guarantee when the `Clock` will stop. To prevent this race condition, `setStopTime` can only be set once on a *Started* `Clock`. A `StopTimeSetError` is thrown if `setStopTime` is called and the *media stop−time* has already been set.

There are no restrictions on calling `setStopTime` on a *Stopped* `Clock`; the stop time can always be reset if the `Clock` is *Stopped*.

Version:
1.42, 97/08/25
See Also:
TimeBase, Player

Variable Index

o **RESET**
Returned by `getStopTime` if the stop−time is unset.

Method Index

o **getMediaNanoseconds**()
Get this `Clock's` current *media time* in nanoseconds.
o **getMediaTime**()
Get this `Clock's` current *media time*.
o **getRate**()
Get the current temporal scale factor.
o **getStopTime**()
Get the last value successfully set by `setStopTime`.
o **getSyncTime**()
Get the current *media time* or the time until this `Clock` will synchronize to its `TimeBase`.
o **getTimeBase**()
Get the `TimeBase` that this `Clock` is using.
o **mapToTimeBase**(Time)
Get the `TimeBase` time corresponding to the specified *media time*.
o **setMediaTime**(Time)
Set the `Clock's` *media time*.
o **setRate**(float)
Set the temporal scale factor.
o **setStopTime**(Time)
Set the *media time* at which you want the `Clock` to stop.

o **setTimeBase**(TimeBase)
Set the `TimeBase` for this `Clock`.
o **stop**()
Stop the `Clock`.
o **syncStart**(Time)
Synchronize the current *media time* to the specified *time−base time* and start the `Clock`.

Variables

o **RESET**

```
public static final Time RESET
```

Returned by `getStopTime` if the stop−time is unset.

Methods

o **setTimeBase**

```
public abstract void setTimeBase(TimeBase master) throws IncompatibleTimeBaseException
```

Set the `TimeBase` for this `Clock`. This method can only be called on a *Stopped* `Clock`. A `ClockStartedError` is thrown if `setTimeBase` is called on a *Started* `Clock`.

A `Clock` has a default `TimeBase` that is determined by the implementation. To reset a `Clock` to its default `TimeBase`, call `setTimeBase(null)`.

Parameters:
master − The new `TimeBase` or `null` to reset the `Clock` to its default `TimeBase`.
Throws: IncompatibleTimeBaseException
Thrown if the `Clock` can't use the specified `TimeBase`.

o **syncStart**

```
public abstract void syncStart(Time at)
```

Synchronize the current *media time* to the specified *time−base time* and start the `Clock`. The `syncStart` method sets the *time−base start−time*, and puts the `Clock` in the *Started* state. This method can only be called on a *Stopped* `Clock`. A `ClockStartedError` is thrown if `setTimeBase` is called on a *Started* `Clock`.

Parameters:
at − The *time−base time* to equate with the current *media time*.

o **stop**

```
public abstract void stop()
```

Stop the `Clock`. Calling `stop` releases the `Clock` from synchronization with the `TimeBase`. After this request is issued, the `Clock` is in the *Stopped* state. If `stop` is called on a *Stopped* `Clock`, the request is ignored.

o **setStopTime**

```
public abstract void setStopTime(Time stopTime)
```

Set the *media time* at which you want the `Clock` to stop. The `Clock` will stop when its *media time* passes the stop−time. To clear the stop time, set it to: `Clock.RESET`.

You can always call `setStopTime` on a *Stopped* `Clock`.

On a *Started* `Clock`, the stop−time can only be set *once*. A `StopTimeSetError` is thrown if `setStopTime` is called and the *media stop−time* has already been set.

Parameters:
stopTime − The time at which you want the `Clock` to stop, in *media time*.

o **getStopTime**

```
public abstract Time getStopTime()
```

Get the last value successfully set by `setStopTime`. Returns the constant `Clock.RESET` if no stop time is set. (`Clock.RESET` is the default stop time.)

Returns:
The current stop time.

o **setMediaTime**

```
public abstract void setMediaTime(Time now)
```

Set the `Clock's` *media time*. This method can only be called on a *Stopped* `Clock`. A `ClockStartedError` is thrown if `setMediaTime` is called on a *Started* `Clock`.

Parameters:
now − The new media time.

o **getMediaTime**

```
public abstract Time getMediaTime()
```

Get this `Clock's` current *media time*. A *Started* `Clock's` *media time* is based on its `TimeBase` and rate, as described in *Starting a Clock*.

Returns:
The current *media time*.

o **getMediaNanoseconds**

```
public abstract long getMediaNanoseconds()
```

Get this Clock's current *media time* in nanoseconds.

Returns:
The current *media time* in nanoseconds.

o **getSyncTime**

```
public abstract Time getSyncTime()
```

Get the current *media time* or the time until this Clock will synchronize to its TimeBase. The getSyncTime method is used by Players and advanced applet writers to synchronize Clocks.

Like getMediaTime, this method returns the Clock's current *media time*, which is based on its TimeBase and rate. However, when syncStart is used to start the Clock, getSyncTime performs a countdown to the time−base start−time, returning the time remaining until the *time−base start−time*. Once the TimeBase reaches the *time−base start−time*, getSyncTime and getMediaTime will return the same value.

o **getTimeBase**

```
public abstract TimeBase getTimeBase()
```

Get the TimeBase that this Clock is using.

o **mapToTimeBase**

```
public abstract Time mapToTimeBase(Time t) throws ClockStoppedException
```

Get the TimeBase time corresponding to the specified *media time*.

Parameters:
t − The *media time* to map from.
Returns:
The *time−base time* in *media−time* coordinates.
Throws: ClockStoppedException
Thrown if mapToTimeBase is called on a *Stopped* Clock.

o **getRate**

```
public abstract float getRate()
```

Get the current temporal scale factor. The scale factor defines the relationship between the Clock's *media time* and its TimeBase.

For example, a rate of 2.0 indicates that *media time* will pass twice as fast as the TimeBase time once the Clock starts. Similarly, a negative rate indicates that the Clock runs in the opposite direction of its TimeBase. All Clocks are guaranteed to support a rate of 1.0, the default rate. Clocks are not required to support any other rate.

o **setRate**

```
public abstract float setRate(float factor)
```

Set the temporal scale factor. The argument *suggests* the scale factor to use.

The setRate method returns the actual rate set by the Clock. Clocks should set their rate as close to the requested value as possible, but are not required to set the rate to the exact value of any argument other than 1.0. A Clock is only guaranteed to set its rate exactly to 1.0.

You can only call this method on a *Stopped* Clock. A ClockStartedError is thrown if setRate is called on a *Started* Clock.

Parameters:
factor − The temporal scale factor (rate) to set.
Returns:
The actual rate set.

Class javax.media.ClockStartedError

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
                   |
                   +----javax.media.MediaError
                           |
                           +----javax.media.ClockStartedError
```

public class **ClockStartedError**
extends MediaError

`ClockStartedError` is thrown by a *Started* `Clock` when a method is invoked that is not legal on a `Clock` in the *Started* state. For example, this error is thrown if `syncStart` or `setTimeBase` is invoked on a *Started* `Clock`. `ClockStartedError` is also thrown if `addController` is invoked on a *Started* `Player`.

Version:
1.15, 97/08/23.
See Also:
Player, Controller, Clock

Constructor Index

o **ClockStartedError**()
Construct a `ClockStartedError` with no message.
o **ClockStartedError**(String)
Construct a `ClockStartedError` that contains the specified reason message.

Constructors

o **ClockStartedError**

```
public ClockStartedError(String reason)
```

Construct a `ClockStartedError` that contains the specified reason message.

o **ClockStartedError**

```
public ClockStartedError()
```

Construct a `ClockStartedError` with no message.

Class javax.media.ClockStoppedException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----javax.media.MediaException
                           |
                           +----javax.media.ClockStoppedException
```

---

public class **ClockStoppedException**
extends MediaException

A `ClockStoppedException` is thrown when a method that expects the *Clock* to be *Started* is called on a *Stopped* `Clock`. For example, this exception is thrown if `mapToTimeBase` is called on a *Stopped* `Clock`.

Version:
1.12, 97/08/23

Constructor Index

o **ClockStoppedException**()
o **ClockStoppedException**(String)

Constructors

o **ClockStoppedException**

```
public ClockStoppedException()
```

o **ClockStoppedException**

```
public ClockStoppedException(String reason)
```

Class javax.media.ConnectionErrorEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.ControllerClosedEvent
                   |
                   +----javax.media.ControllerErrorEvent
                           |
                           +----javax.media.ConnectionErrorEvent
```

---

public class **ConnectionErrorEvent**
extends ControllerErrorEvent

A `ConnectionErrorEvent` is posted when an error occurs within a `DataSource` when obtaining data or communicating with a server.

Version:
1.6, 97/08/23

Constructor Index

o **ConnectionErrorEvent**(Controller)
o **ConnectionErrorEvent**(Controller, String)

Constructors

o **ConnectionErrorEvent**

```
public ConnectionErrorEvent(Controller from)
```

o **ConnectionErrorEvent**

```
public ConnectionErrorEvent(Controller from,
                            String why)
```

All Packages Class Hierarchy This Package Previous Next Index

---

Interface javax.media.Control

public interface **Control**

The base interface for processing `Control` objects.

Version:
1.13, 97/08/26

Method Index

o **getControlComponent**()
Get the `Component` associated with this `Control` object.

Methods

o **getControlComponent**

```
public abstract Component getControlComponent()
```

Get the `Component` associated with this `Control` object. For example, this method might return a slider for volume control or a panel containing radio buttons for CODEC control. The `getControlComponent` method can return `null` if there is no GUI control for this `Control`.

Interface javax.media.Controller

public interface **Controller**
extends Clock, Duration

`Controller`, which extends `Clock`, provides resource−allocation state information, event generation, and a mechanism for obtaining objects that provide additional control over a `Controller`.

Controller life−cycle

As a `Clock`, a `Controller` is always either *Started* or *Stopped*. However, `Controller` subdivides `Clock's` *Stopped* state into five resource−allocation phases: *Unrealized*, *Realizing*, *Realized*, *Prefetching*, and *Prefetched*.

The motivation for these life−cycle states is to provide programmatic control over potentially time−consuming operations. For example, when a `Controller` is first constructed, it's in the *Unrealized* state. While *Realizing*, the `Controller` performs the communication necessary to locate all of the resources it needs to function (such as communicating with a server, other controllers, or a file system). The `realize` method allows an application to initiate this potentially time−consuming process (*Realizing*) at an appropriate time. When a `Controller` is *Realizing* or *Prefetching*, it will eventually transition to another state, such as *Realized*, *Prefetched*, or even *Unrealized*.

Because a `Controller` is often in one state on its way to another, its destination or *target* state is an integral part of the `Controller` life−cycle. You can query a `Controller` to determine both its current state and its target state.

A `Controller` typically moves from the *Unrealized* state through *Realizing* to the *Realized* state, then through *Prefetching* to the *Prefetched* state, and finally on to the *Started* state. When a `Controller` finishes because the end of the media stream is reached, its stop time is reached, or the `stop` method is invoked, the `Controller` moves from the *Started* state back to *Prefetched* or possibly back to *Realized*, ready to repeat the cycle.

To use a `Controller`, you set up parameters to manage its movement through these life−cycle states and then move it through the states using the `Controller` state transition methods. To keep track of the `Controller's` current state, you monitor the state transition events that it posts when changing states.

State Transition Methods

A `Controller` has five methods that are used to induce life−cycle state changes: `realize`, `prefetch`, `deallocate`, `syncStart`, and `stop`. To transition a `Controller` to the *Realized*, *Prefetched*, or *Started* state, you use the corresponding method: `realize`, `prefetch`, or `syncStart`. The `deallocate` and `stop` methods can change a requested state transition or trigger a state change.

The forward transition methods (`realize`, `prefetch`, and `syncStart`) are executed asynchronously and return immediately. When the requested operation is complete, the `Controller` posts a `ControllerEvent` that indicates that the target state has been reached, `stop` or `deallocate` has been invoked, or that an error occurred.

The `deallocate`, and `stop` methods can change the target state and induce a transition back to a previous state. For example, calling `deallocate` on a `Controller` in the *Prefetching* state will move it back to *Realized*. These methods are synchronous.

State Transition Events

A `Controller` often moves between states in an asynchronous manner. To facilitate the tracking of a `Controller's` state, every time its state or target state changes, the `Controller` is required to post a `TransitionEvent` that describes its previous state, current state, and new target state. By monitoring the `Controller` event stream, you can determine exactly what a `Controller` is doing at any point in time.

When one of the asynchronous forward state transition methods completes, the `Controller` posts the appropriate `TransitionEvent` or a `ControllerErrorEvent` indicating that the `Controller` is no longer usable. For more information about `ControllerEvents`, see the ***Controller Events section***.

To facilitate simple asynchronous method protocols, a `Controller` always posts a method completion event when one of the asynchronous forward state transition methods is invoked, even if no state or target state change occurs. For example, if `realize` is called on a *Prefetching* `Controller`, a `RealizeCompleteEvent` is immediately posted, even though the `Controller` remains in the *Prefetching* state and the target state is still *Prefetched*. The method completion events always report the `Controller's` previous, current, and target state at the time the event was posted.

Controller States

This section describes the semantics of each of the `Controller` states.

Unrealized State

A newly instanced `Controller` starts in the *Unrealized* state. An *Unrealized* `Controller` knows very little about its internals and does not have enough information to acquire all of the resources it needs to function. In particular, an *Unrealized* `Controller` does not know enough to properly construct a `Clock`. Therefore, it is illegal to call the following methods on an *Unrealized* `Controller`:

- `getTimeBase`
- `setTimeBase`
- `setMediaTime`
- `setRate`
- `setStopTime`
- `getStartLatency`

A `NotRealizedError` is thrown if any of these methods are called on an *Unrealized* `Controller`.

Realizing and Realized States

A `Controller` is *Realized* when it has obtained all of the information necessary for it to acquire the resources it needs to function. A *Realizing* `Controller` is in the process of identifying the resources that it needs to acquire. *Realizing* can be a resource and time–consuming process. A *Realizing* `Controller` might have to communicate with a server, read a file, or interact with a set of other objects.

Although a *Realized* `Controller` does not have to acquire any resources, a *Realized* `Controller` is likely to have acquired all of the resources it needs except those that imply exclusive use of a scarce system resource, such as an audio device or MPEG decoding hardware.

Normally, a `Controller` moves from the *Unrealized* state through *Realizing* and on to the *Realized* state. After `realize` has been invoked on a `Controller`, the only way it can return to the *Unrealized* state is if `deallocate` is invoked before *Realizing* completes. Once a `Controller` reaches the *Realized* state, it never returns to the *Unrealized* state; it remains in one of four states: *Realized*, *Prefetching*, *Prefetched*, or *Started*.

Realize method

The `realize` method executes asynchronously and completion is signaled by a `RealizeCompleteEvent` or a `ControllerErrorEvent`.

Prefetching and Prefetched States

Once *Realized*, a `Controller` might still need to perform a number of time–consuming tasks before it is ready to be started. For example, it might need to acquire scarce hardware resources, fill buffers with media data, or perform other start–up processing. While performing these tasks, the `Controller` is in the *Prefetching* state. When finished, it moves into the *Prefetched* state. Over a `Controller's` lifetime, *Prefetching* might have to recur when certain methods are invoked. For example, calling `setMediaTime` might cause a `Player` to be *Prefetched* again before it is *Started*.

Once a `Controller` is *Prefetched*, it is capable of starting as quickly as is possible for that `Controller`. *Prefetching* reduces the startup latency of a `Controller` to the minimum possible value. (The startup latency is the value returned by `getStartLatency`.)

Typically, a `Controller` moves from the *Realized* state through *Prefetching* and on to the *Prefetched* state. Once *Prefetched*, a `Controller` remains *Prefetched* unless `deallocate`, `syncStart` or a method that changes its state and increases its startup latency is invoked, such as `setMediaTime`.

A *Started* `Controller` returns to the *Prefetched* or *Realized* state when it stops.

Prefetch Method

The `prefetch` method is asynchronous and its completion is signaled by a `PrefetchCompleteEvent` or a `ControllerErrorEvent`. As a convenience, if `prefetch` is invoked before a `Controller` has reached the *Realized* state, an implicit `realize` is invoked by changing the target state to *Prefetched*. Both a `RealizeCompleteEvent` and a `PrefetchCompleteEvent` are posted by the `Controller` as it transitions to the *Prefetched* state.

If a `Controller` is *Prefetching* and cannot obtain all of the resources it needs to start, it posts a `ResourceUnavailableEvent` instead of a `PrefetchCompleteEvent`. This is a catastrophic error condition from which the `Controller` cannot recover.

Started State

Once *Prefetched*, a `Controller` can enter the *Started* state. A *Started* `Controller's` `Clock` is running and it is processing data. A `Controller` returns to the *Prefetched* or *Realized* state when it stops because it has reached its stop time, reached the end of the media, or because the `stop` method was invoked.

When the `Controller` moves from the *Prefetched* to the *Started* state, it posts a `StartEvent`. When it moves from the *Started* state to a stopped state, it posts a `StopEvent`.

A `Controller` is a `Clock`; therefore, `syncStart`, `setTimeBase`, `setMediaTime`, and `setRate` are illegal when the `Controller` is in the *Started* state.

syncStart

The only way to start a `Controller` is to call `syncStart`.

It is illegal to call `syncStart` unless the `Controller` is in the *Prefetched* state. If `syncStart` is called before the `Controller` is *Prefetched*, a `NotPrefetchedError` is thrown. `Player` defines a `start` method that relaxes this requirement.

Freeing the Resources Used by a Controller

`Deallocate` is used to stop a `Controller's` resource consumption. For example, when `Applet.stop` is called, `deallocate` should be called to free the resources that the `Controller` was using. `Deallocate` stops any resource–consuming activity and releases any exclusive–use resources that the `Controller` has acquired. `Deallocate` executes synchronously; when `deallocate` returns, the resources have been released.

If the `Controller` is *Unrealized* or *Realizing*, calling `deallocate` returns it to the *Unrealized* state. Otherwise, calling `deallocate` returns a `Controller` to the *Realized* state. Regardless of the state that a `Controller` is in, `deallocate` must relinquish any exclusive–use system resources that it holds; the only way to guarantee that a `Controller` is not holding resources is to call the `deallocate` method.

It is illegal to call `deallocate` on a *Started* `Controller`. You must stop the `Controller` before it can relinquish its resources.

When `deallocate` is called, a `Controller` posts a special `StopEvent`, `DeallocateEvent`.

Controller Events

Controller events asynchronously deliver information about Controller state changes. There are four kinds of notifications: life–cycle transition, method acknowledgement, state notification, and error notification.

To receive events, an object must implement the ControllerListener interface and use the addControllerListener method to register its interest in a Controller's events. All Controller events are posted to each registered listener.

The Controller event mechanism is extensible and some Controllers define events other than the ones described here. For example, the DurationUpdateEvents that a Player posts are ControllerEvents .

TransitionEvent
: TransitionEvents are posted when a Controller's current or target state changes. TransitionEvent is subclassed to provide a small set of events that are posted for particular kinds of transitions that merit special interest. The class name of the event indicates either the reason that the event was posted (such as EndOfMediaEvent), or the particular transition that the event represents (such as PrefetchCompleteEvent).

  In addition to being posted for state transitions, the method acknowledgement events RealizeCompleteEvent, PrefetchCompleteEvent, StartEvent, DeallocateEvent, and StopByRequestEvent are always posted to signify method completion even if no transition has taken place.

  RealizeCompleteEvent
  : Posted when a Controller moves from *Realizing* to the *Realized* state, or when the realize method is invoked and the Controller is already *Realized*.

  PrefetchCompleteEvent
  : Posted when a Controller moves from *Prefetching* to the *Prefetched* state, or when the prefetch method is invoked and the Controller is already *Prefetched*.

  StartEvent
  : Posted when a Controller moves from *Prefetched* to *Started*.

  StopEvent
  : Posted when a Controller moves backward. For example, when moving from *Prefetched* to *Realized* or from *Started* to *Prefetched*. The *reason* that a stop event occurs is often important; this information is provided through several subclasses of StopEvent.

    StopAtTimeEvent
    : Posted when a Controller changes state because it has reached its stop time.

    StopByRequestEvent
    : Posted when a Controller changes state because stop is invoked. This event is also posted as an acknowledgement to stop requests.

    DeallocateEvent
    : Posted when the deallocate method is invoked, indicating a possible state change and the loss of exclusive–use resources. The current state is either *Unrealized* or *Realized*. This event doesn't always indicate a state change. For example, it is posted even if deallocate is called on a *Realized* Controller.

    EndOfMediaEvent
    : Posted when a Controller has reached the end of the media.

    ControllerClosedEvent
    : When a Controller closes it is no longer usable, and it will post a ControllerClosedEvent. Once this has happened method calls on the Controller have undefined behavior. A Controller will close for one of two reasons. Either the close method was invoked on the Controller, or an error has occurred. If a Controller is closed because the close method was invoked, it posts a ControllerClosedEvent. If an error occurs it posts one of the ControllerErrorEvents.

ControllerErrorEvent
: This is the super class of all of the error events that can be posted by a Controller. While this event is rarely posted, you should watch for it when processing other error events––this is how you can detect implementation–specific error events.

  When a ControllerErrorEvent is posted, it indicates a catastrophic error from which the Controller cannot recover. There is no recovery mechanism for a Controller once one of these events has been posted.

  ResourceUnavailableEvent
  : This error event is posted during *Prefetching* or *Realizing* to indicate that the operation has failed because a required resource was unavailable.

  DataLostErrorEvent
  : This error event is posted when a Controller has lost data.

  InternalErrorEvent
  : This error event is posted when something goes wrong with the Controller for an implementation–specific reason. This usually indicates that there is a problem with the implementation.

Status Change Events
: A small number of status changes occur in a Controller where notification of the change is useful, particularly for updating user interface components. Notification of these changes is provided through three ControllerEvents:

  RateChangeEvent
  : Posted when the rate of a Controller changes.

  StopTimeChangeEvent
  : Posted when the stop time of a Controller changes.

  MediaTimeSetEvent
  : Posted when the media time has been set using the setMediaTime method. This event is *not* periodically posted as media time changes due to normal Controller processing and Clock operation.

Controls

A Control is an object that provides a way to affect some aspect of a Controller's operation in a specific way. The Control interface provides access to a GUI Component that is specific to the particular Control. For example, the GainControl interface provides a way to display a GUI control that allows the user to change the volume.

A `Controller` makes a collection of `Controls` available that effect the `Controller's` behavior. To access these `Controls`, you use the `getControls` method, which returns an array of supported `Controls`. If you know the full class or interface name of the `Control` you want, you can use `getControl`.

Since an application using a `Controller` might not know how to use all of the `Controls` that a `Controller` supports, it can make the functionality available to a user by providing access to the `Component` for the `Control`.

Version:
1.63, 97/08/28
See Also:
Player, Control, ControllerListener, ControllerEvent, TransitionEvent, RealizeCompleteEvent, PrefetchCompleteEvent, StartEvent, StopEvent, EndOfMediaEvent, ControllerErrorEvent, DataLostErrorEvent, ResourceUnavailableEvent, InternalErrorEvent, RateChangeEvent, MediaTimeSetEvent, ClockStartedError, NotRealizedError

---

Variable Index

o **LATENCY_UNKNOWN**
Returned by `getStartLatency`.
o **Prefetched**
Returned by `getState`.
o **Prefetching**
Returned by `getState`.
o **Realized**
Returned by `getState`.
o **Realizing**
Returned by `getState`.
o **Started**
Returned by getState.
o **Unrealized**
Returned by `getState`.

Method Index

o **addControllerListener**(ControllerListener)
Specify a `ControllerListener` to which this `Controller` will send events.
o **close**()
Release all resources and cease all activity.
o **deallocate**()
Abort the current operation and cease any activity that consumes system resources.
o **getControl**(String)
Get the `Control` that supports the class or interface specified.
o **getControls**()
Get a list of the `Control` objects that this `Controller` supports.
o **getStartLatency**()
Get the `Controller's` start latency in nanoseconds.
o **getState**()
Get the current state of this `Controller`.
o **getTargetState**()
Get the current target state of this `Controller`.
o **prefetch**()
Process as much data as necessary to reduce the `Controller's` start latency to the shortest possible time.
o **realize**()
Construct the media dependent portions of the `Controller`.
o **removeControllerListener**(ControllerListener)
Remove the specified listener from this `Controller's` listener list.

Variables

o **LATENCY_UNKNOWN**

```
public static final Time LATENCY_UNKNOWN
```

Returned by `getStartLatency`.

o **Unrealized**

```
public static final int Unrealized
```

Returned by `getState`.

o **Realizing**

```
public static final int Realizing
```

Returned by `getState`.

o **Realized**

```
public static final int Realized
```

Returned by `getState`.

o **Prefetching**

```
public static final int Prefetching
```

Returned by `getState`.

o **Prefetched**

```
public static final int Prefetched
```

Returned by `getState`.

o **Started**

```
public static final int Started
```

Returned by getState.

Methods

o **getState**

```
public abstract int getState()
```

Get the current state of this `Controller`. The state is an integer constant as defined above.

Note: A race condition can occur between the return of this method and the execution of a state changing method.

Returns:
The `Controller's` current state.

o **getTargetState**

```
public abstract int getTargetState()
```

Get the current target state of this `Controller`. The state is an integer constant as defined above.

Note: A race condition can occur between the return of this method and the execution of a state changing method.

Returns:
The `Controller's` current target state.

o **realize**

```
public abstract void realize()
```

Construct the media dependent portions of the `Controller`. This can require examining media data and might take some time to complete.

The `realize` method puts the `Controller` into the *Realizing* state and returns immediately. When `realize` is complete and the `Controller` is in the *Realized* state, the `Controller` posts a `RealizeCompleteEvent`.

o **prefetch**

```
public abstract void prefetch()
```

Process as much data as necessary to reduce the `Controller's` start latency to the shortest possible time. This typically requires examining media data and takes some time to complete.

The `prefetch` method puts the `Controller` into the *Prefetching* state and returns immediately. When *Prefetching* is complete and the `Controller` is in the *Prefetched* state, the `Controller` posts a `PrefetchCompleteEvent`.

o **deallocate**

```
public abstract void deallocate()
```

Abort the current operation and cease any activity that consumes system resources. If a `Controller` is not yet *Realized*, it returns to the *Unrealized* state. Otherwise, the `Controller` returns to the *Realized* state.

It is illegal to call `deallocate` on a *Started* `Controller`. A `ClockStartedError` is thrown if `deallocate` is called and the `Controller` is in the *Started* state.

o **close**

```
public abstract void close()
```

Release all resources and cease all activity. The `close` method indicates that the `Controller` will no longer be used, and the `Controller` can shut itself down. A `ControllerClosedEvent` is posted. Methods invoked on a closed `Controller` might throw errors.

o **getStartLatency**

```
public abstract Time getStartLatency()
```

Get the `Controller's` start latency in nanoseconds. The start latency represents a worst–case estimate of the amount of time it will take to present the first frame of data.

This method is useful for determining how far in advance the `syncStart` method must be invoked to ensure that media will be rendered at the specified start time.

For a `Controller` that has a variable start latency, the value returned represents the maximum possible start latency. If you call `getStartLatency` on a `Controller` that isn't *Prefetched* and `getStartLatency` returns `LATENCY_UNKNOWN`, calling `prefetch` and then calling `getStartLatency` again after the `Controller` posts a `PrefetchCompleteEvent` might return a more accurate estimate. If `getStartLatency` still returns `LATENCY_UNKNOWN`, the start latency is indeterminate and you might not be able to use `syncStart` to synchronize the `Controller` with other `Controllers`.

Note: In most cases, the value returned by `getStartLatency` will change once the `Controller` is *Prefetched*.

Returns:
The time it will take before the first frame of media can be presented.

o **getControls**

```
public abstract Control[] getControls()
```

Get a list of the `Control` objects that this `Controller` supports. If there are no controls, an array of length zero is returned.

Returns:
A list of `Controller` `Controls`.

o **getControl**

```
public abstract Control getControl(String forName)
```

Get the `Control` that supports the class or interface specified. The full class or interface name should be specified. `Null` is returned if the `Control` is not supported.

Returns:
`Control` for the class or interface name.

o **addControllerListener**

```
public abstract void addControllerListener(ControllerListener listener)
```

Specify a `ControllerListener` to which this `Controller` will send events. A `Controller` can have multiple `ControllerListeners`.

Parameters:
listener – The listener to which the `Controller` will post events.

o **removeControllerListener**

```
public abstract void removeControllerListener(ControllerListener listener)
```

Remove the specified listener from this `Controller's` listener list.

Parameters:
listener – The listener that has been receiving events from this `Controller`.

Class javax.media.ControllerClosedEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.ControllerClosedEvent
```

public class **ControllerClosedEvent**
extends ControllerEvent

A `ControllerClosedEvent` describes an event that is generated when an a `Controller` is closed. This implies that the `Controller` is no longer operational.

Version:
1.6, 97/08/23.
See Also:
Controller, ControllerListener

Variable Index

o **message**

Constructor Index

o **ControllerClosedEvent**(Controller)
Construct a `ControllerClosedEvent`.
o **ControllerClosedEvent**(Controller, String)

Method Index

o **getMessage**()
Obtain the message describing why this event occurred.

Variables

o **message**

```
protected String message
```

Constructors

o **ControllerClosedEvent**

```
public ControllerClosedEvent(Controller from)
```

Construct a `ControllerClosedEvent`.

o **ControllerClosedEvent**

```
public ControllerClosedEvent(Controller from,
                             String why)
```

Methods

o **getMessage**

```
public String getMessage()
```

Obtain the message describing why this event occurred.

Returns:
Message describing event cause.

Class javax.media.ControllerErrorEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.ControllerClosedEvent
                   |
                   +----javax.media.ControllerErrorEvent
```

public class **ControllerErrorEvent**
extends ControllerClosedEvent

A `ControllerErrorEvent` describes an event that is generated when an error condition occurs that will cause a `Controller` to cease functioning. Events should only subclass from `ControllerErrorEvent` if the error being reported will result in catastrophic failure if action is I not taken, or if the `Controller` has already failed. A `ControllerErrorEvent` indicates that the `Controller` is closed.

Version:
1.16, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **ControllerErrorEvent**(Controller)
o **ControllerErrorEvent**(Controller, String)

Constructors

o **ControllerErrorEvent**

```
public ControllerErrorEvent(Controller from)
```

o **ControllerErrorEvent**

```
public ControllerErrorEvent(Controller from,
                            String why)
```

Class javax.media.ControllerEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
```

---

public class **ControllerEvent**
extends Object
implements MediaEvent

`ControllerEvent` is the base class for events generated by a `Controller`. These events are used by `ControllerListener`.

Java Beans Compatibility

This class is designed to support the Java Beans event model. In order to enable

Version:
1.11, 97/08/25
See Also:
Controller, ControllerListener, MediaEvent

Constructor Index

o **ControllerEvent**(Controller)

Method Index

o **getSource**()
o **getSourceController**()
Get the `Controller` that posted this event.

Constructors

o **ControllerEvent**

```
public ControllerEvent(Controller from)
```

Methods

o **getSourceController**

```
public Controller getSourceController()
```

Get the `Controller` that posted this event. The returned `Controller` has at least one active listener. (The `addListener` method has been called on the `Controller`).

Returns:
The `Controller` that posted this event.

o **getSource**

```
public Object getSource()
```

Interface javax.media.ControllerListener

public interface **ControllerListener**

`ControllerListener` is an interface for handling asynchronous events generated by `Controllers`.

Java Beans Support

Any implementation of this objext is required to be subclassed from either java.util.EventListener or sunw.util.EventListener.

Version:
1.18, 97/08/25
See Also:
Controller

Method Index

o **controllerUpdate**(ControllerEvent)
This method is called when an event is generated by a `Controller` that this listener is registered with.

Methods

o **controllerUpdate**

```
public abstract void controllerUpdate(ControllerEvent event)
```

This method is called when an event is generated by a `Controller` that this listener is registered with.

Parameters:
event – The event generated.

Class javax.media.DataStarvedEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.DataStarvedEvent
```

---

public class **DataStarvedEvent**
extends StopEvent

`DataStarvedEvent` indicates that a `Controller` has lost data or has stopped receiving data altogether. This transitions the `Controller` into a *Stopped* state.

Version:
1.17, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **DataStarvedEvent**(Controller, int, int, int, Time)

Constructors

o **DataStarvedEvent**

```
public DataStarvedEvent(Controller from,
                        int previous,
                        int current,
                        int target,
                        Time mediaTime)
```

Class javax.media.DeallocateEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.DeallocateEvent
```

public class **DeallocateEvent**
extends StopEvent

A `DeallocateEvent` is posted as an acknowledgement of the invocation of the `deallocate` method. It implies that the scarce resources associated with this `Controller` are no longer available and must be reacquired.

A `DeallocateEvent` can be posted at any time regardless of the `Controller's` previous or current state. `DeallocateEvent` is a `StopEvent` because if the `Controller` is in the *Started* state when the event is posted, it transitions to one of the *Stopped* states.

Version:
1.11, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **DeallocateEvent**(Controller, int, int, int, Time)

Constructors

o **DeallocateEvent**

```
public DeallocateEvent(Controller from,
                       int previous,
                       int current,
                       int target,
                       Time mediaTime)
```

Interface javax.media.Duration

public interface **Duration**

The `Duration` interface provides a way to determine the duration of the media being played by a media object. Media objects that expose a media duration implement this interface.

A `Controller` that supports the `Duration` interface posts a `DurationUpdateEvent` whenever its duration changes.

Version:
1.16, 97/08/23
See Also:
Controller, DurationUpdateEvent

Variable Index

o **DURATION_UNBOUNDED**
Returned by `getDuration`.
o **DURATION_UNKNOWN**
Returned by `getDuration`.

Method Index

o **getDuration**()
Get the duration of the media represented by this object.

Variables

o **DURATION_UNBOUNDED**

```
public static final Time DURATION_UNBOUNDED
```

Returned by `getDuration`.

o **DURATION_UNKNOWN**

```
public static final Time DURATION_UNKNOWN
```

Returned by `getDuration`.

Methods

o **getDuration**

```
public abstract Time getDuration()
```

Get the duration of the media represented by this object. The value returned is the media's duration when played at the default rate. If the duration can't be determined (for example, the media object is presenting live video) `getDuration` returns `DURATION_UNKNOWN`.

Returns:
A `Time` object representing the duration or DURATION_UNKNOWN.

Class javax.media.DurationUpdateEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.DurationUpdateEvent
```

public class **DurationUpdateEvent**
extends ControllerEvent

`DurationUpdateEvent` is posted by a `Controller` when its duration changes.

Version:
1.10, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **DurationUpdateEvent**(Controller, Time)

Method Index

o **getDuration**()
Get the duration of the media that this `Controller` is using.

Constructors

o **DurationUpdateEvent**

```
public DurationUpdateEvent(Controller from,
                           Time newDuration)
```

Methods

o **getDuration**

```
public Time getDuration()
```

Get the duration of the media that this `Controller` is using.

Returns:
The duration of this `Controller's` media.

Class javax.media.EndOfMediaEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.EndOfMediaEvent
```

---

public class **EndOfMediaEvent**
extends StopEvent

An `EndOfMediaEvent` indicates that the `Controller` has reached the end of its media and is stopping.

Version:
1.21, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **EndOfMediaEvent**(Controller, int, int, int, Time)

Constructors

o **EndOfMediaEvent**

```
public EndOfMediaEvent(Controller from,
                       int previous,
                       int current,
                       int target,
                       Time mediaTime)
```

Class javax.media.GainChangeEvent

```
java.lang.Object
   |
   +----javax.media.GainChangeEvent
```

---

public class **GainChangeEvent**
extends Object
implements MediaEvent

A `GainChangeEvent` is posted by a `GainControl` when its state has been updated.

Java Beans support

Any implementation of this object is required to be subclassed from either java.util.EventObject or sunw.util.EventObject.

Version:
1.14, 97/08/26
See Also:
GainControl, GainChangeListener

Constructor Index

o **GainChangeEvent**(GainControl, boolean, float, float)

Method Index

o **getDB**()
Get the `GainControl's` new gain value in dB.
o **getLevel**()
Get the `GainControl's` new gain value in the level scale.
o **getMute**()
Get the `GainControl's` new mute value.
o **getSource**()
Get the object that posted this event.
o **getSourceGainControl**()
Get the `GainControl` that posted this event.

Constructors

o **GainChangeEvent**

```
public GainChangeEvent(GainControl from,
                       boolean mute,
                       float dB,
                       float level)
```

Methods

o **getSource**

```
public Object getSource()
```

Get the object that posted this event.

Returns:
The object that posted this event.

o **getSourceGainControl**

```
public GainControl getSourceGainControl()
```

Get the `GainControl` that posted this event.

Returns:
The `GainControl` that posted this event.

o **getDB**

```
public float getDB()
```

Get the `GainControl's` new gain value in dB.

Returns:
The `GainControl's` new gain value, in dB.

o **getLevel**

```
public float getLevel()
```

Get the `GainControl's` new gain value in the level scale.

Returns:
The `GainControl's` new gain, in the level scale.

o **getMute**

```
public boolean getMute()
```

Get the `GainControl's` new mute value.

Returns:
The `GainControl's` new mute value.

Interface javax.media.GainChangeListener

public interface **GainChangeListener**

`GainChangeListener` is an interface for handling `GainChangeEvents` generated by `GainControls`.

Java Beans support

It is required that any implementation of this object is sub−classed either from java.util.EventListener, or sunw.util.EventListener.

Version:
1.11, 97/08/25.
See Also:
GainControl, GainChangeEvent

Method Index

o **gainChange**(GainChangeEvent)
This method is called to deliver a `GainChangeEvent` when the state of a `GainControl` changes.

Methods

o **gainChange**

```
public abstract void gainChange(GainChangeEvent event)
```

This method is called to deliver a `GainChangeEvent` when the state of a `GainControl` changes.

Parameters:
event − The event generated.

Interface javax.media.GainControl

public interface **GainControl**
extends Control

`GainControl` is an interface for manipulating audio signal gain.

Gain and Gain Measures

Gain is a multiplicative value applied to an audio signal that modifies the amplitude of the signal. This interface allows the gain to be specified in either decibels or using a floating point value that varies between 0.0 and 1.0.

Specifying Gain in Decibels

The decibel scale is valid over all `float` values. A gain of 0.0 dB implies that the audio signal is neither amplified nor attenuated. Positive values amplify the audio signal, negative values attenuate the audio signal. The relationship between a linear gain multiplier and the gain specified in decibels is:

```
value = pow(10.0, gainDB/20.0)
```

Specifying Gain in the Level Scale

The level scale ranges from 0.0 to 1.0, where 0.0 represents a gain that is virtually indistinguishable from silence and 1.0 represents the value that is, in some sense, the maximum gain. In other words, 1.0 represents the highest gain value that produces "useful" results. The mapping for producing a linear multiplicative value is implementation dependent.

Decibel and Level Interactions

The dB and level scales are representations of the same gain value. Calling `setLevel` will affect subsequent `getDB` invocations. Level and dB are interrelated in the following ways:

- Level Silence Threshold. After `setLevel(0.0)`, `getDB` returns the value for which smaller values are not usefully distinguishable from silence. Calling `setDB` with values equal to or less than this silence threshold causes `getLevel` to return a value of 0.0.
- Level Maximum Threshold. After `setLevel(1.0)`, `getDB` returns the value for which larger values are not useful. Calling `setDB` with values equal to or greater than this threshold causes `getLevel` to return a value of 1.0.
- The decibel interface is not limited to the thresholds described by the level interface. For example, if you call `setDB` with a value that is greater than the maximum level threshold and then immediately call `getDB`, `getDB` returns the gain that was returned by the `setDB`, *not* the value that would be returned if you called `setLevel(1.0)` and then called `getDB`.
- Both measures increase gain monotonically with increasing measure values.

Defaults

Gain defaults to a value of 0.0 dB. The corresponding level is implementation dependent. Note that for some implementations, the default level might change on a per−instance basis.

Mute

Muting is independent of the gain. If mute is `true`, no audio signal is produced by this object; if mute is `false` an audio signal is produced and the gain is applied to the signal.

Gain Change Events

When the state of the `GainControl` changes, a `GainChangeEvent` is posted. This event is delivered through an object that implements `GainChangeListener` and has been registered as a listener with the `GainControl` using `addGainChangeListener`.

Version:
1.33, 97/08/23
See Also:
GainChangeEvent, GainChangeListener, Control

Method Index

o **addGainChangeListener**(GainChangeListener)
Register for gain change update events.
o **getDB**()
Get the current gain set for this object in dB.
o **getLevel**()
Get the current gain set for this object as a value between 0.0 and 1.0
o **getMute**()
Get the mute state of the signal associated with this `GainControl`.
o **removeGainChangeListener**(GainChangeListener)
Remove interest in gain change update events.
o **setDB**(float)
Set the gain in decibels.
o **setLevel**(float)
Set the gain using a floating point scale with values between 0.0 and 1.0.
o **setMute**(boolean)
Mute or unmute the signal associated with this `GainControl`.

Methods

o **setMute**

```
public abstract void setMute(boolean mute)
```

Mute or unmute the signal associated with this `GainControl`. Calling `setMute(true)` on an object that is already muted is ignored, as is calling `setMute(false)` on an object that is not currently muted. Going from a muted to an unmuted state doesn't effect the gain.

Parameters:
mute – Specify `true` to mute the signal, `false` to unmute the signal.

o **getMute**

```
public abstract boolean getMute()
```

Get the mute state of the signal associated with this `GainControl`.

Returns:
The mute state.

o **setDB**

```
public abstract float setDB(float gain)
```

Set the gain in decibels. Setting the gain to 0.0 (the default) implies that the audio signal is neither amplified nor attenuated. Positive values amplify the audio signal and negative values attenuate the signal.

Parameters:
gain – The new gain in dB.
Returns:
The gain that was actually set.

o **getDB**

```
public abstract float getDB()
```

Get the current gain set for this object in dB.

Returns:
The gain in dB.

o **setLevel**

```
public abstract float setLevel(float level)
```

Set the gain using a floating point scale with values between 0.0 and 1.0. 0.0 is silence; 1.0 is the loudest useful level that this `GainControl` supports.

Parameters:
level – The new gain value specified in the level scale.
Returns:
The level that was actually set.

o **getLevel**

```
public abstract float getLevel()
```

Get the current gain set for this object as a value between 0.0 and 1.0

Returns:
The gain in the level scale (0.0–1.0).

o **addGainChangeListener**

```
public abstract void addGainChangeListener(GainChangeListener listener)
```

Register for gain change update events. A `GainChangeEvent` is posted when the state of the `GainControl` changes.

Parameters:
listener – The object to deliver events to.

o **removeGainChangeListener**

```
public abstract void removeGainChangeListener(GainChangeListener listener)
```

Remove interest in gain change update events.

Parameters:
listener – The object that has been receiving events.

Class javax.media.IncompatibleSourceException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----javax.media.MediaException
                           |
                           +----javax.media.IncompatibleSourceException
```

public class **IncompatibleSourceException**
extends MediaException

An `IncompatibleSourceException` is thrown by a `MediaHandler` when `setSource` is invoked and the `MediaHandler` cannot support the `DataSource`.

Version:
1.2, 97/08/23.
See Also:
DataSource, MediaHandler, Manager

Constructor Index

o **IncompatibleSourceException**()
o **IncompatibleSourceException**(String)

Constructors

o **IncompatibleSourceException**

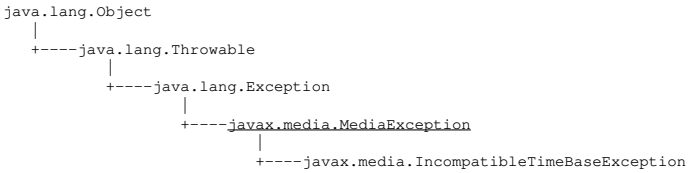
```
public IncompatibleSourceException()
```

o **IncompatibleSourceException**

```
public IncompatibleSourceException(String reason)
```

All Packages Class Hierarchy This Package Previous Next Index

Class javax.media.IncompatibleTimeBaseException



public class **IncompatibleTimeBaseException**
extends MediaException

An `IncompatibleTimeBaseException` is generated when `Clock.setTimeBase` is invoked using a `TimeBase` that the `Clock` cannot support. This happens for certain types of `Players` that can only be driven by their own internal clocks, such as certain commercial video servers.

Note: A `Player` might throw this exception when `addController` is called because of the implied `setTimeBase` in `addController`.

Version:
1.9, 97/08/23.
See Also:
Clock, Player

Constructor Index

o **IncompatibleTimeBaseException**()
o **IncompatibleTimeBaseException**(String)

Constructors

o **IncompatibleTimeBaseException**

```
public IncompatibleTimeBaseException()
```

o **IncompatibleTimeBaseException**

```
public IncompatibleTimeBaseException(String reason)
```

Class javax.media.InternalErrorEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.ControllerClosedEvent
                   |
                   +----javax.media.ControllerErrorEvent
                           |
                           +----javax.media.InternalErrorEvent
```

public class **InternalErrorEvent**
extends ControllerErrorEvent

An `InternalErrorEvent` indicates that a `Controller` failed for implementation−specific reasons. This event indicates that there are problems with the implementation of the `Controller`.

Version:
1.7, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **InternalErrorEvent**(Controller)
o **InternalErrorEvent**(Controller, String)

Constructors

o **InternalErrorEvent**

```
public InternalErrorEvent(Controller from)
```

o **InternalErrorEvent**

```
public InternalErrorEvent(Controller from,
                          String message)
```

All Packages Class Hierarchy This Package Previous Next Index

---

Class javax.media.Manager

```
java.lang.Object
   |
   +----javax.media.Manager
```

---

public final class **Manager**
extends Object

`Manager` is the access point for obtaining system dependent resources such as `Players`, `DataSources`, and the system `TimeBase`.

A `Player` is an object used to control and render multimedia data that is specific to the content type of the data. A `DataSource` is an object used to deliver time−based multimedia data that is specific to a delivery protocol. A `DataSource` provides a `Player` with media data; a `Player` must have a `DataSource`. `Manager` provides access to a protocol and media independent mechanism for constructing `Players` and `DataSources`.

Creating Players and DataSources

`Manager` will create`Players` from a `URL`, a `MediaLocator` or a `DataSource`. Creating a `Player` requires the following:

- Obtain the connected `DataSource` for the specified protocol
- Obtain the `Player` for the content−type specified by the `DataSource`
- Attach the `DataSource` to the `Player` using the `setSource` method.

Finding DataSources by Protocol

A `MediaLocator` defines a protocol for obtaining content. `DataSources` are identified by the protocol that they support. `Manager` uses the protocol name to find `DataSource` classes.

To find a `DataSource` using a `MediaLocator`, `Manager` constructs a list of class names from the protocol package−prefix list and the protocol name obtained from the `MediaLocator`. For each class name in the constructed list a new `DataSource` is instanced, the `MediaLocator` is attached, and the `DataSource` is connected. If no errors have occurred, the procces is considered finished and the connected `DataSource` is used by `Manager` in any following operations. If there was an error then the next class name in the list is tried. The exact details of the search algorithm is described in the method documentation below.

Finding Players by Content Type

A `Player` is a `MediaHandler`. A `MediaHandler` is a an object that reads data from a `DataSource`. There are two types of supported `MediaHandler`: `MediaProxy`, and `Player`.

`MediaHandlers` are identified by the content type that they support. A `DataSource` identifies the content type of the data it produces with the `getContentType` method. `Manager` uses the content type name to find instances of `MediaHandler`.

To find a `MediaHandler` using a content type name, `Manager` constructs a list of class names from the content package−prefix list and the content type name. For each class name in the constructed list a new `MediaHandler` is instanced, and the `DataSource` is attached to the `MediaHandler` using MediaHandler.setSource.

If the `MediaHandler` is a `Player` and the `setSource` was successful the process is finished and the `Player` is returned. If the `setSource` failed, another name in the list is tried.

If the `MediaHandler` is a `MediaProxy` then a new `DataSource` is obtained from the `MediaProxy`, a new list is created for the content type the `DataSource` supports and the whole thing is tried again.

If a valid `Player`, is not found then the whole procedure is repeated is repeated with "unknown" substituted for the content−type name. The "unknown" content type is supported by generic `Players` that are capable of handling a large variety of media types, often in a platform dependent way.

The detailed creation algorithm is specified in the methods below.

Player Threads

`Players` render media data asynchronously from the main program flow. This implies that a `Player` must often manage one or more threads. The threads managed by the `Player` are not in the thread group of the application that calls `createPlayer`.

System Time Base

All `Players` need a `TimeBase`. Many use a system−wide `TimeBase`, often based on a time−of−day clock. `Manager` provides access to the system `TimeBase` through `getSystemTimeBase`.

Version:
1.57, 97/08/28.
See Also:
URL, MediaLocator, PackageManager, DataSource, URLDataSource, MediaHandler, Player, MediaProxy, TimeBase

---

Variable Index

o **UNKNOWN_CONTENT_NAME**

Method Index

o **createDataSource**(MediaLocator)
Create a DataSource for the specified media.
o **createDataSource**(URL)
Create a DataSource for the specified media.
o **createPlayer**(DataSource)
Create a Player for the DataSource.
o **createPlayer**(MediaLocator)
Create a Player for the specified media.
o **createPlayer**(URL)
Create a Player for the specified media.
o **getDataSourceList**(String)
Build a list of DataSource class names from the protocol prefix–list and a protocol name.
o **getHandlerClassList**(String)
Build a list of Handler/CODE> classes from the content-prefix-list and a content name.
o **getSystemTimeBase**()
Get the time-base object for the system.

Variables

o **UNKNOWN_CONTENT_NAME**

```
public static final String UNKNOWN_CONTENT_NAME
```

Methods

o **createPlayer**

```
public static Player createPlayer(URL sourceURL) throws IOException, NoPlayerException
```

Create a Player for the specified media. This creates a MediaLocator from the URL and then calls createPlayer.

Parameters:
sourceURL – The URL that describes the media data.
Returns:
A new Player.
Throws: NoPlayerException
Thrown if no Player can be found.
Throws: IOException
Thrown if there was a problem connecting with the source.

o **createPlayer**

```
public static Player createPlayer(MediaLocator sourceLocator) throws IOException, NoPlayerException
```

Create a Player for the specified media.

The algorithm for creating a Player from a MediaLocator is:

1. Get the protocol from the MediaLocator.
2. Get a list of DataSource classes that support the protocol, using the protocol package–prefix–list.
3. For each source class in the list:
    1. Instantiate a new DataSource,
    2. Call the connect method to connect the source.
    3. Get the media content–type–name (using getContentType) from the source.
    4. Get a list of MediaHandler classes that support the media–content–type–name, using the content package–prefix–list.
    5. For each MediaHandler class in the list:
        1. Instantiate a new MediaHandler.
        2. Attach the source to the MediaHandler by calling MediaHandler.setSource.
        3. If there are no failures, determine the type of the MediaHandler; otherwise try the next MediaHandler in the list.
        4. If the MediaHandler is a Player, return the new Player.
        5. If the MediaHandler is a MediaProxy, obtain a new DataSource from the MediaProxy, obtain the list of MediaHandlers that support the new DataSource, and continue searching the new list.
    6. If no MediaHandler is found for this source, try the next source in the list.
4. If no Player is found after trying all of the sources, reuse the source list.
   This time, for each source class in the list:
    1. Instantiate the source.
    2. Call the connect method to connect to the source.
    3. Use the content package–prefix–list to create a list of MediaHandler classes that support the "unknown" content–type–name.
    4. For each MediaHandler class in the list, search for a Player as in the previous search.
        1. If no Player is found after trying all of the sources, a NoPlayerException is thrown.

Parameters:
sourceLocator – A MediaLocator that describes the media content.
Returns:
A Player for the media described by the source.
Throws: NoPlayerException
Thrown if no Player can be found.
Throws: IOException
Thrown if there was a problem connecting with the source.

o **createPlayer**

```
public static Player createPlayer(DataSource source) throws IOException, NoPlayerException
```

Create a `Player` for the `DataSource`.

The algorithm for creating a `Player` from a `DataSource` is:

1. Get the media content−type−name from the source by calling `getContentType`.
2. Use the content package−prefix−list to get a list of `Player` classes that support the media content−type name.
3. For each `Player` class in the list:
    1. Instantiate a new `Player`.
    2. Attach the source to the `Player` by calling `setSource` on the `Player`.
    3. If there are no failures, return the new `Player`; otherwise, try the next `Player` in the list.
4. If no `Player` is found for this source:
    1. Use the content package−prefix−list to create a list of `Player` classes that support the "unknown" content−type−name.
    2. For each `Player` class in the list:
        1. Instantiate a new `Player`.
        2. Attach the source to the `Player` by calling `setSource` on the `Player`.
        3. If there are no failures, return the new `Player`; otherwise, try the next `Player` in the list.
5. If no `Player` can be created, a `NoPlayerException` is thrown.

Parameters:
DataSource − The `DataSource` that describes the media content.
Returns:
A new `Player`.
Throws: NoPlayerException
Thrown if a `Player` can't be created.
Throws: IOException
Thrown if there was a problem connecting with the source.

- **createDataSource**

```
public static DataSource createDataSource(URL sourceURL) throws IOException, NoDataSourceException
```

Create a `DataSource` for the specified media.

Parameters:
sourceURL − The `URL` that describes the media data.
Returns:
A new `DataSource` for the media.
Throws: NoDataSourceException
Thrown if no `DataSource` can be found.
Throws: IOException
Thrown if there was a problem connecting with the source.

- **createDataSource**

```
public static DataSource createDataSource(MediaLocator sourceLocator) throws IOException, NoDataSourceException
```

Create a `DataSource` for the specified media.

Returns a data source for the protocol specified by the `MediaLocator`. The returned data source is *connected*; `DataSource.connect` has been invoked.

The algorithm for creating a `DataSource` from a `MediaLocator` is:

1. Get the protocol from the `MediaLocator`.
2. Use the protocol package−prefix list to get a list of `DataSource` classes that support the protocol.
3. For each source class in the list:
    1. Instantiate a new `DataSource`.
    2. Call `connect` to connect the source.
    3. If there are no errors, return the connected source; otherwise, try the next source in the list.
4. If no source has been found, obtain a `URL` from the `MediaLocator` and use it to create a `URLDataSource`
5. If no source can be found, a `NoDataSourceException` is thrown.

Parameters:
sourceLocator − The source protocol for the media data.
Returns:
A connected `DataSource`.
Throws: NoDataSourceException
Thrown if no `DataSource` can be found.
Throws: IOException
Thrown if there was a problem connecting with the source.

- **getSystemTimeBase**

```
public static TimeBase getSystemTimeBase()
```

Get the time−base object for the system.

Returns:
The system time base.

- **getDataSourceList**

```
public static Vector getDataSourceList(String protocolName)
```

Build a list of `DataSource` class names from the protocol prefix−list and a protocol name.

The first name in the list will always be:

```
media.protocol.<protocol>DataSource
```

Each additional name looks like:

```
<protocol-prefix>.media.protocol.<protocol>.DataSource
```

for every <protocol-prefix> in the protocol-prefix-list.

Parameters:
protocol - The name of the protocol the source must support.
Returns:
A vector of strings, where each string is a `Player` class-name.

- **getHandlerClassList**

```
public static Vector getHandlerClassList(String contentName)
```

Build a list of `Handler/CODE> classes from the content-prefix-list and a content name.`

`The first name in the list will always be:`

```
media.content.<contentType>.Handler
```

Each additional name looks like:

```
<content-prefix>.media.content.<contentName>.Player
```

for every <content-prefix> in the content-prefix-list.

Parameters:
contentName - The content type to use in the class name.
Returns:
A vector of strings where each one is a `Player` class-name.

Class javax.media.MediaError

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
                   |
                   +----javax.media.MediaError
```

---

public class **MediaError**
extends Error

A `MediaError` indicates an error condition that occurred through incorrect usage of the API. You should not check for `MediaErrors`.

Version:
1.11, 97/08/23.

Constructor Index

o **MediaError**()
o **MediaError**(String)

Constructors

o **MediaError**

```
public MediaError()
```

o **MediaError**

```
public MediaError(String reason)
```

Interface javax.media.MediaEvent

public interface **MediaEvent**

`MediaEvent` is the base interface for events supported by the media framework.

Java Beans support

In order to support the Java Beans event model an implementation of MediaEvent is required to sub−class java.util.EventObject. If an implementation is designed to support the 1.0.2 JDK then it may alternatively sub−class sunw.util.EventObject to provide the support appropriate support. **Any class that subclasses** `MediaEvent` **must resolve to either java.util.EventObject or sunw.util.EventObject.**

Version:
1.3, 97/08/25.
See Also:
ControllerEvent, GainChangeEvent

Method Index

o getSource()

Methods

o getSource

```
public abstract Object getSource()
```

Class javax.media.MediaException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----javax.media.MediaException
```

---

public class **MediaException**
extends Exception

A `MediaException` indicates an unexpected error condition in a JavaMedia method.

Version:
1.9, 97/08/28

Constructor Index

o **MediaException**()
o **MediaException**(String)

Constructors

o **MediaException**

```
public MediaException()
```

o **MediaException**

```
public MediaException(String reason)
```

Interface javax.media.MediaHandler

public interface **MediaHandler**

`MediaHandler` is the base interface for objects that read and manage media content delivered from a `DataSource`.

There are currently two supported types of `MediaHandler/code>: Player and MediaProxy.`

Version:
`1.4, 97/08/23.`
See Also:
`Player`, `MediaProxy`

Method Index

o **setSource**(DataSource)
Set the media source the MediaHandler should use to obtain content.

Methods

o **setSource**

```
public abstract void setSource(DataSource source) throws IOException, IncompatibleSourceException
```

Set the media source the `MediaHandler` should use to obtain content.

Parameters:
source – The `DataSource` used by this `MediaHandler`.
Throws: IOException
Thrown if there is an error using the `DataSource`
Throws: IncompatibleSourceException
Thrown if this `MediaHandler` cannot make use of the `DataSource`.

Class javax.media.MediaLocator

```
java.lang.Object
   |
   +----javax.media.MediaLocator
```

---

public class **MediaLocator**
extends Object

`MediaLocator` describes the location of media content. `MediaLocator` is closely related to `URL`. `URL`s can be obtained from `MediaLocators`, and `MediaLocators` can be constructed from `URL`. Unlike a `URL`, a `MediaLocator` can be instanced without a `URLStreamHandler` installed on the System.

Version:
1.8, 97/08/25.
See Also:
URL, URLStreamHandler

Constructor Index

o **MediaLocator**(String)
o **MediaLocator**(URL)

Method Index

o **getProtocol**()
Get the beginning of the locator string up to but not including the first colon.
o **getRemainder**()
Get the `MediaLocator` string with the protocol removed.
o **getURL**()
Get the `URL` associated with this `MediaLocator`.
o **toExternalForm**()
Create a string from the `URL` argument that can be used to construct the `MediaLocator`.
o **toString**()
Used for printing `MediaLocators`.

Constructors

o **MediaLocator**

```
public MediaLocator(URL url)
```

Parameters:
url – The `URL` to construct this media locator from.

o **MediaLocator**

```
public MediaLocator(String locatorString)
```

Methods

o **getURL**

```
public URL getURL() throws MalformedURLException
```

Get the `URL` associated with this `MediaLocator`.

o **getProtocol**

```
public String getProtocol()
```

Get the beginning of the locator string up to but not including the first colon.

Returns:
The protocol for this `MediaLocator`.

o **getRemainder**

```
public String getRemainder()
```

Get the `MediaLocator` string with the protocol removed.

Returns:
The argument string.

o **toString**

```
public String toString()
```

Used for printing `MediaLocators`.

Returns:
A string for printing `MediaLocators`.
Overrides:

toString in class Object

o **toExternalForm**

```
public String toExternalForm()
```

Create a string from the `URL` argument that can be used to construct the `MediaLocator`.

Returns:
A string for the `MediaLocator`.

Interface javax.media.MediaProxy

public interface **MediaProxy**
extends MediaHandler

`MediaProxy` is a `MediaHandler` which processes content from one `DataSource`, to produce another `DataSource`.

Typically, a `MediaProxy` reads a text configuration file that contains all of the information needed to make a connection to a server and obtain media data. To produce a `Player` from a `MediaLocator` referencing the configuration file, `Manger`:

- constructs a `DataSource` for the protocol described by the `MediaLocator`
- constructs a `MediaProxy` to read the configuration file using the content−type of the `DataSource`
- obtains a new `DataSource` from the `MediaProxy`
- constructs the `Player` using the content−type of the new `DataSource`

Version:
1.10, 97/08/25.
See Also:
Manager

Method Index

o **getDataSource**()
Obtain the new `DataSource`.

Methods

o **getDataSource**

```
public abstract DataSource getDataSource() throws IOException, NoDataSourceException
```

Obtain the new `DataSource`. The `DataSource` is already connected.

Returns:
the new `DataSource` for this content.
Throws: IOException
Thrown when if there are IO problems in reading the the original or new `DataSource`.
Throws: NoDataSourceException
Thrown if this proxy can't produce a `DataSource`.

Class javax.media.MediaTimeSetEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.MediaTimeSetEvent
```

public class **MediaTimeSetEvent**
extends ControllerEvent

A `MediaTimeSetEvent` is posted by a `Controller` when its media−time has been set with the `setMediaTime` method.

Version:
1.13, MediaTimeSetEvent.java.
See Also:
Controller, ControllerListener

Constructor Index

o **MediaTimeSetEvent**(Controller, Time)

Method Index

o **getMediaTime**()
Get the new media time of the `Controller` that generated this event.

Constructors

o **MediaTimeSetEvent**

```
public MediaTimeSetEvent(Controller from,
                         Time newMediaTime)
```

Methods

o **getMediaTime**

```
public Time getMediaTime()
```

Get the new media time of the `Controller` that generated this event.

Returns:
The `Controller's` new media time.

Class javax.media.NoDataSourceException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----javax.media.MediaException
                           |
                           +----javax.media.NoDataSourceException
```

---

public class **NoDataSourceException**
extends MediaException

A `NoDataSourceException` is thrown when a `DataSource` can't be found for a particular `URL` or `MediaLocator`.

Version:
1.8, 97/08/23.

Constructor Index

o **NoDataSourceException**()
o **NoDataSourceException**(String)

Constructors

o **NoDataSourceException**

```
public NoDataSourceException()
```

o **NoDataSourceException**

```
public NoDataSourceException(String reason)
```

Class javax.media.NoPlayerException

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Exception
                   |
                   +----javax.media.MediaException
                           |
                           +----javax.media.NoPlayerException
```

---

public class **NoPlayerException**
extends MediaException

A `NoPlayerException` is thrown when a `PlayerFactory` can't find a `Player` for a particular `URL` or `MediaLocator`.

Version:
1.8, 97/08/23.

Constructor Index

o **NoPlayerException**()
o **NoPlayerException**(String)

Constructors

o **NoPlayerException**

```
public NoPlayerException()
```

o **NoPlayerException**

```
public NoPlayerException(String reason)
```

Class javax.media.NotPrefetchedError

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
                   |
                   +----javax.media.MediaError
                           |
                           +----javax.media.NotPrefetchedError
```

---

public class **NotPrefetchedError**
extends MediaError

`NotPrefetchedError` is thrown when a method that requires a `Controller` to be in the *Prefetched* state is called and the `Controller` has not been *Prefetched*.

This typically happens when `syncStart` is invoked on a *Stopped* `Controller` that hasn't been *Prefetched*.

Version:
1.12, 97/08/23.
See Also:
Controller

Constructor Index

o **NotPrefetchedError**(String)

Constructors

o **NotPrefetchedError**

```
public NotPrefetchedError(String reason)
```

Class javax.media.NotRealizedError

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
                   |
                   +----javax.media.MediaError
                           |
                           +----javax.media.NotRealizedError
```

---

public class **NotRealizedError**
extends MediaError

`NotRealizedError` is thrown when a method that requires a `Controller` to be in the *Realized* state is called and the `Controller` is not *Realized*.

For example, this can happen when `getComponents` is called on an *Unrealized* `Player`.

Version:
1.8, 97/08/23.
See Also:
Controller, Player

Constructor Index

o **NotRealizedError**(String)

Constructors

o **NotRealizedError**

```
public NotRealizedError(String reason)
```

Class javax.media.PackageManager

```
java.lang.Object
   |
   +----javax.media.PackageManager
```

---

public class **PackageManager**
extends Object

A `PackageManager` maintains a persistent store of package−prefix lists. A package prefix specifies the prefix for a complete class name. A factory uses a package−prefix list to find a class that might belong to any of the packages that are referenced in the prefix list.

The `Manager` uses package−prefix lists to find protocol handlers and content handlers for time−based media.

The current version of a package−prefix list is obtained with the `get<package-prefix>List` method. This method returns the prefix list in use; any changes to the list take effect immediately. Unless it is made persistent with `commit<package-prefix>List`, a package−prefix list is only valid while the `Manager` is referenced. The `commit<package-prefix>List` method ensures that any changes made to a package−prefix list are still visible the next time that the `Manager` is referenced.

Version:
1.11, 97/08/23.
See Also:
Manager

Constructor Index

o **PackageManager**()

Method Index

o **commitContentPrefixList**()
Make changes to the content prefix−list persistent.
o **commitProtocolPrefixList**()
Make changes to the protocol package−prefix list persistent.
o **getContentPrefixList**()
Get the current value of the content package−prefix list.
o **getProtocolPrefixList**()
Get the current value of the protocol package−prefix list.
o **setContentPrefixList**(Vector)
Set the current value of the content package−prefix list.
o **setProtocolPrefixList**(Vector)
Set the protocol package−prefix list.

Constructors

o **PackageManager**

```
public PackageManager()
```

Methods

o **getProtocolPrefixList**

```
public static Vector getProtocolPrefixList()
```

Get the current value of the protocol package−prefix list.

Returns:
The protocol package−prefix list.

o **setProtocolPrefixList**

```
public static void setProtocolPrefixList(Vector list)
```

Set the protocol package−prefix list. This is required for changes to take effect.

Parameters:
list − The new package−prefix list to use.

o **commitProtocolPrefixList**

```
public static void commitProtocolPrefixList()
```

Make changes to the protocol package−prefix list persistent.

This method throws a `SecurityException` if the calling thread does not have access to system properties.

o **getContentPrefixList**

```
public static Vector getContentPrefixList()
```

Get the current value of the content package−prefix list. Any changes made to this list take effect immediately.

Returns:
The content package−prefix list.

o **setContentPrefixList**

```
public static void setContentPrefixList(Vector list)
```

Set the current value of the content package−prefix list. This is required for changes to take effect.

Parameters:
list − The content package−prefix list to set.

o **commitContentPrefixList**

```
public static void commitContentPrefixList()
```

Make changes to the content prefix−list persistent.

This method throws a `SecurityException` if the calling thread does not have access to system properties.

Interface javax.media.Player

public interface **Player**
extends MediaHandler, Controller, Duration

`Player` is a `MediaHandler` for rendering and controlling time based media data. `Player` extends both the `Controller` and `Duration` interfaces `Player` provides methods for obtaining AWT components, media processing controls, and a way to manage other `Controllers`.

How a Player Differs from a Controller

`Player` relaxes some restrictions that a `Controller` imposes on what methods can be called on a *Started*, *Stopped*, or *Unrealized* `Controller`. It also provides a way to manage groups of `Controllers`.

Methods Restricted to *Stopped* Players

The following methods can only be called on a `Player` in one of the *Stopped* states. If they are invoked on a *Started* `Player`, a `ClockStartedError` is thrown.

- `setTimeBase`
- `syncStart`
- `deallocate`
- `addController`
- `removeController`

Methods Allowed on *Started* Players

Unlike a `Controller`, the following methods are *legal* on a `Player` in the *Started* state:

- `setMediaTime`
- `setRate`

Invoking these methods on a *Started* `Player` might initiate significant and time–consuming processing, depending on the location and type of media being processed. These methods might also cause the state of the `Player` to change. If this happens, the appropriate `TransitionEvents` are posted by the `Player` when its state changes.

For example, a `Player` might have to enter the *Prefetching* state to process a `setMediaTime` invocation. In this case, the `Player` posts a `RestartingEvent`, a `PrefetchCompleteEvent`, and a `StartEvent` as it moves from the *Started* state to *Prefetching*, back to *Prefetched*, and finally back to the *Started* state.

Methods that are Illegal on *Unrealized* Players

As with `Controller`, it is illegal to call the following methods on an *Unrealized* `Player`:

- `getTimeBase`
- `setTimeBase`
- `setMediaTime`
- `setRate`
- `setStopTime`
- `getStartLatency`

It is also illegal to call the following `Player` methods on an *Unrealized* `Player`:

- `getVisualComponent`
- `getControlPanelComponent`
- `getGainControl`
- `addController`
- `removeController`

The `Player` throws a `NotRealizedError` if any of these methods are called while the `Player` is in the *Unrealized* state.

Start Method

As a convenience, `Player` provides a `start` method that can be invoked before a `Player` is *Prefetched*. This method attempts to transition the `Player` to the *Started* state from whatever state it's currently in. For example, if the `Player` is *Unrealized*, `start` implicitly calls `realize`, `prefetch`, and `Clock.syncStart`. The appropriate `TransitionEvents` are posted as the `Player` moves through each state on its way to *Started*.

RestartingEvent

If `setMediaTime` or `setRate` cause a perceptible delay in the presentation of the media, the `Player` posts a `RestartingEvent` and transitions to the *Prefetching* state. The previous state and target state of a `RestartingEvent` is always *Started*. `RestartingEvent` is a subclass of `StopEvent`.

DurationUpdateEvent

Because a `Player` cannot always know the duration of the media it is playing, the `Duration` interface defines that `getDuration` returns `Duration.DURATION_UNKNOWN` until the duration can be determined. A `DurationUpdateEvent` is generated when the `Player` can determine its duration or the if its duration changes, which can happen at any time. When the end of the media is reached, the duration should be known.

Managing other Controllers

In some situations, an application might want to use a single `Player` to control other `Players` or `Controllers`. A single controlling `Player` can be used to invoke `start`, `stop`, `setMediaTime`, and

other methods on the entire group. The controlling Player manages all of the state transitions and event posting.

It is also possible to construct a simple Controller to update animations, report on media time–line progress, or provide other timing–related functions. Such Controllers can operate in sync with a controlling Player.

Adding a Controller

To have a Player assume control over a Controller, use the addController method. A Controller can only be added to a *Stopped* Player. If addController is called on a *Started* Player, a ClockStartedError is thrown. An *Unrealized* Controller cannot be added to a Player; a NotRealizedError is thrown if the Controller is *Unrealized*.

Once a Controller has been added, the Player:

- Invokes setTimeBase on the Controller with the Player's TimeBase. If this fails, addController throws an IncompatibleTimeBaseException.
- Synchronizes the Controller with the Player using setMediaTime, setStopTime, and setRate.
- Takes the added Controller's latency into account when computing the Player's start latency. When getStartLatency is called, the Player returns the greater of: its latency before the Controller was added and the latency of the added Controller.
- Takes the added Controller's duration into account when computing the Player's duration. When getDuration is called, the Player returns the greater of: its duration before the Controller was added and the duration of the added Controller. If either of these values is DURATION_UNKNOWN, getDuration returns DURATION_UNKNOWN. If either of these values is DURATION_UNBOUNDED getDuration returns DURATION_UNBOUNDED.
- Adds itself as a ControllerListener for the added Controller so that it can manage the events that the Controller generates. (See the Events section below for more information.)
- Invokes control methods on the added Controller in response to methods invoked on the Player. The methods that affect managed Controllers are discussed below.

Once a Controller has been added to a Player, methods should only be called on the Controller through the managing Player. It is not defined how the Controller or Player will behave if methods are called directly on an added Controller. You cannot place a controlling Player under the control of a Player that it is managing; the resulting behavior is undefined.

When a Controller is added to a Player, the Player does not transition the added Controller to new state, nor does the Player transition itself forward. The Player either transitions back to the *realized* state if the added Controller is *realized* or *prefetching* or it stays in the *prefetched* state if the both the Player and the added Controller are in the *prefetched* state. If the Player makes a state transition as a result of adding a Controller the Player posts a TransitionEvent.

Removing a Controller

To stop a Player from managing another Controller, call removeController. The managing Player must be *Stopped* before removeController can be called. A ClockStartedError is thrown if removeController is called on a *Started* Player.

When a Controller is removed from a Player's control, the Player:

- Resets the Controller's TimeBase to its default.
- Recalculates its duration and posts a DurationUpdateEvent if the Player's duration is different without the Controller added.
- Recalculates its start latency.

Setting the Media Time and Rate of a Managing Player

When you call setMediaTime on a Player that's managing other Controllers, its actions differ depending on whether or not the Player is *Started*. If the Player is not *Started*, it simply invokes setMediaTime on all of the Controllers it's managing.

If the Player is *Started*, it posts a RestartingEvent and performs the following tasks for each managed Controller:

- Invokes stop on the Controller.
- Invokes setMediaTime on the Controller.
- Invokes prefetch on the Controller.
- Waits for a PrefetchCompleteEvent from the Controller.
- Invokes syncStart on the Controller

The same is true when setRate is called on a managing Player. The Player attempts to set the specified rate on all managed Controllers, stopping and restarting the Controllers if necessary. If some of the Controllers do not support the requested rate, the Player returns the rate that was actually set. All Controllers are guaranteed to have been successfully set to the rate returned.

Starting a Managing Player

When you call start on a managing Player, all of the Controllers managed by the Player are transitioned to the *Prefetched* state. When the Controllers are *Prefetched*, the managing Player calls syncStart with a time consistent with the latencies of each of the managed Controllers.

Calling realize, prefetch, stop, or deallocate on a Managing Player

When you call realize, prefetch, stop, or deallocate on a managing Player, the Player calls that method on all of the Controllers that it is managing. The Player moves from one state to the next when all of its Controllers have reached that state. For example, a Player in the *Prefetching* state does not transition into the *Prefetched* state until all of its managed Controllers are *Prefetched*. The Player posts TransitionEvents normally as it changes state.

Calling syncStart or setStopTime on a Managing Player

When you call `syncStart` or `setStopTime` on a managing `Player`, the `Player` calls that method on all of the `Controllers` that it is managing. (The `Player` must be in the correct state or an error is thrown. For example, the `Player` must be *Prefetched* before you can call `syncStart`.)

Setting the Time Base of a Managing Player

When `setTimeBase` is called on a managing `Player`, the `Player` calls `setTimeBase` on all of the `Controllers` it's managing. If `setTimeBase` fails on any of the `Controllers`, an `IncompatibleTimeBaseException` is thrown and the `TimeBase` last used is restored for all of the `Controllers`.

Getting the Duration of a Managing Player

Calling `getDuration` on a managing `Player` returns the maximum duration of all of the added `Controllers` and the managing `Player`. If the `Player` or any `Controller` has not resolved its duration, `getDuration` returns `Duration.DURATION_UNKNOWN`.

Closing a Managing Player

When `close` is called on a managing `Player` all managed `Controllers` are closed as well.

Events

Most events posted by a managed `Controller` are filtered by the managing `Player`. Certain events are sent directly from the `Controller` through the `Player` and to the listeners registered with the `Player`.

To handle the events that a managed `Controller` can generate, the `Player` registers a listener with the `Controller` when it is added. Other listeners that are registered with the `Controller` must be careful not to invoke methods on the `Controller` while it is being managed by the `Player`. Calling a control method on a managed `Controller` directly will produce unpredictable results.

When a `Controller` is removed from the `Player's` list of managed `Controllers`, the `Player` removes itself from the `Controller's` listener list.

Transition Events

A managing `Player` posts `TransitionEvents` normally as it moves between states, but the managed `Controllers` affect when the `Player` changes state. In general, a `Player` does not post a transition event until all of its managed `Controllers` have posted the event.

Status Change Events

The managing `Player` collects the `RateChangeEvents`, `StopTimeChangeEvents`, and `MediaTimeSetEvents` posted by its managed `Controllers` and posts a single event for the group.

DurationUpdateEvent

A `Player` posts a `DurationUpdateEvent` when it determines its duration or its duration changes. A managing `Player's` duration might change if a managed `Controller` updates or discovers its duration. In general, if a managed `Controller` posts a `DurationUpdateEvent` and the new duration changes the managing `Player's` duration, the `Player` posts a `DurationUpdateEvent`

CachingControlEvent

A managing `Player` reposts `CachingControlEvents` received from a `Players` that it manages, but otherwise ignores the events.

ControllerErrorEvents

A managing `Player` immediately reposts any `ControllerErrorEvent` received from a `Controller` that it is managing. After a `ControllerErrorEvent` has been received from a managed `Controller`, a managing `Player` no longer invokes any methods on the managed `Controller`; the managed `Controller` is ignored from that point on.

Version:
1.75, 97/08/25

See Also:
Manager, GainControl, Clock, TransitionEvent, RestartingEvent, DurationUpdateEvent, Component

---

Method Index

o **addController**(Controller)
Assume control of another `Controller`.
o **getControlPanelComponent**()
Obtain the `Component` that provides the default user interface for controlling this `Player`.
o **getGainControl**()
Obtain the object for controlling this `Player's` audio gain.
o **getVisualComponent**()
Obtain the display `Component` for this `Player`.
o **removeController**(Controller)
Stop controlling a `Controller`.
o **start**()
Start the `Player` as soon as possible.

Methods

o **getVisualComponent**

```
public abstract Component getVisualComponent()
```

Obtain the display `Component` for this `Player`. The display `Component` is where visual media is rendered. If this `Player` has no visual component, `getVisualComponent` returns `null`. For example, `getVisualComponent` might return `null` if the `Player` only plays audio.

Returns:
The media display `Component` for this `Player`.

o **getGainControl**

```
public abstract GainControl getGainControl()
```

Obtain the object for controlling this `Player's` audio gain. If this player does not have a `GainControl`, `getGainControl` returns `null`. For example, `getGainControl` might return `null` if the `Player` does not play audio data.

Returns:
The `GainControl` object for this `Player`.

o **getControlPanelComponent**

```
public abstract Component getControlPanelComponent()
```

Obtain the `Component` that provides the default user interface for controlling this `Player`. If this `Player` has no default control panel, `getControlPanelComponent` returns `null`.

Returns:
The default control panel GUI for this `Player`.

o **start**

```
public abstract void start()
```

Start the `Player` as soon as possible. The `start` method attempts to transition the `Player` to the *Started* state. If the `Player` has not been *Realized* or *Prefetched*, `start` automatically performs those actions. The appropriate events are posted as the `Player` moves through each state.

o **addController**

```
public abstract void addController(Controller newController) throws IncompatibleTimeBaseException
```

Assume control of another `Controller`.

Parameters:
newController – The `Controller` to be managed.
Throws: IncompatibleTimeBaseException
Thrown if the added `Controller` cannot take this * `Player's TimeBase`.

o **removeController**

```
public abstract void removeController(Controller oldController)
```

Stop controlling a `Controller`.

Parameters:
oldController – The `Controller` to stop managing.

Class javax.media.PrefetchCompleteEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.PrefetchCompleteEvent
```

---

public class **PrefetchCompleteEvent**
extends TransitionEvent

A `PrefetchCompleteEvent` is posted when a `Controller` finishes *Prefetching*. This occurs when a `Controller` moves from the *Prefetching* state to the *Prefetched* state, or as an acknowledgement that the `prefetch` method was called and the `Controller` is already *Prefetched*.

Version:
1.20, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **PrefetchCompleteEvent**(Controller, int, int, int)

Constructors

o **PrefetchCompleteEvent**

```
public PrefetchCompleteEvent(Controller from,
                             int previous,
                             int current,
                             int target)
```

Class javax.media.RateChangeEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.RateChangeEvent
```

---

public class **RateChangeEvent**
extends ControllerEvent

A `RateChangeEvent` is a `ControllerEvent` that is posted when a `Controller's` rate changes.

Version:
1.11, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **RateChangeEvent**(Controller, float)

Method Index

o **getRate**()
Get the new rate of the `Controller` that generated this event.

Constructors

o **RateChangeEvent**

```
public RateChangeEvent(Controller from,
                       float newRate)
```

Methods

o **getRate**

```
public float getRate()
```

Get the new rate of the `Controller` that generated this event.

Returns:
The `Controller's` new rate.

Class javax.media.RealizeCompleteEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.RealizeCompleteEvent
```

---

public class **RealizeCompleteEvent**
extends TransitionEvent

A `RealizeCompleteEvent` is posted when a `Controller` finishes *Realizing*. This occurs when a `Controller` moves from the *Realizing* state to the *Realized* state, or as an acknowledgement that the `realize` method was called and the `Controller` is already *Realized*.

Version:
1.14, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **RealizeCompleteEvent**(Controller, int, int, int)

Constructors

o **RealizeCompleteEvent**

```
public RealizeCompleteEvent(Controller from,
                            int previous,
                            int current,
                            int target)
```

Class javax.media.ResourceUnavailableEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.ControllerClosedEvent
                   |
                   +----javax.media.ControllerErrorEvent
                           |
                           +----javax.media.ResourceUnavailableEvent
```

public class **ResourceUnavailableEvent**
extends ControllerErrorEvent

A `ResourceUnavailableEvent` indicates that a `Controller` was unable to allocate a resource that it requires for operation.

Version:
1.21, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **ResourceUnavailableEvent**(Controller)
o **ResourceUnavailableEvent**(Controller, String)

Constructors

o **ResourceUnavailableEvent**

```
public ResourceUnavailableEvent(Controller from)
```

o **ResourceUnavailableEvent**

```
public ResourceUnavailableEvent(Controller from,
                                String message)
```

All Packages Class Hierarchy This Package Previous Next Index

Class javax.media.RestartingEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.RestartingEvent
```

public class **RestartingEvent**
extends StopEvent

A `RestartingEvent` indicates that a `Controller` has moved from the *Started* state back to the *Prefetching* state (a *Stopped* state) and intends to return to the *Started* state when *Prefetching* is complete. This occurs when a *Started* `Player` is asked to change its rate or media time and to fulfill the request must prefetch its media again.

Version:
1.14, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **RestartingEvent**(Controller, int, int, int, Time)

Constructors

o **RestartingEvent**

```
public RestartingEvent(Controller from,
                       int previous,
                       int current,
                       int target,
                       Time mediaTime)
```

Class javax.media.StartEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StartEvent
```

public class **StartEvent**
extends TransitionEvent

`StartEvent` is a `TransitionEvent` that indicates that a `Controller` has entered the *Started* state. Entering the *Started* state implies that `syncStart` has been invoked, providing a new *media time* to *time−base time* mapping. `StartEvent` provides the *time−base time* and the *media−time* that *Started* this `Controller`.

Version:
1.31, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **StartEvent**(Controller, int, int, int, Time, Time)
Construct a new `StartEvent`.

Method Index

o **getMediaTime**()
Get the clock time (*media time*) when the `Controller` started.
o **getTimeBaseTime**()
Get the time−base time that started the `Controller`.

Constructors

o **StartEvent**

```
public StartEvent(Controller from,
                  int previous,
                  int current,
                  int target,
                  Time mediaTime,
                  Time tbTime)
```

Construct a new `StartEvent`. The `from` argument identifies the `Controller` that is generating this event. The `mediaTime` and the `tbTime` identify the *media−time* to *time−base−time* mapping that *Started* the `Controller`

Parameters:
from − The `Controller` that has *Started*.
mediaTime − The media time when the `Controller`*Started*.
tbTime − The time−base time when the `Controller`*Started*.

Methods

o **getMediaTime**

```
public Time getMediaTime()
```

Get the clock time (*media time*) when the `Controller` started.

Returns:
The `Controller's` *media time* when it started.

o **getTimeBaseTime**

```
public Time getTimeBaseTime()
```

Get the time−base time that started the `Controller`.

Returns:
The *time−base time* associated with the `Controller` when it started.

Class javax.media.StopAtTimeEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.StopAtTimeEvent
```

---

public class **StopAtTimeEvent**
extends StopEvent

A `StopAtTimeEvent` indicates that the `Controller` has stopped because it reached its stop time.

Version:
1.11, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **StopAtTimeEvent**(Controller, int, int, int, Time)

Constructors

o **StopAtTimeEvent**

```
public StopAtTimeEvent(Controller from,
                       int previous,
                       int current,
                       int target,
                       Time mediaTime)
```

Class javax.media.StopByRequestEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
                           |
                           +----javax.media.StopByRequestEvent
```

---

public class **StopByRequestEvent**
extends StopEvent

A `StopByRequestEvent` indicates that the `Controller` has stopped in response to a `stop` call. This event is posted as an acknowledgement even if the `Controller` is already *Stopped*.

Version:
1.11, 97/08/23.
See Also:
Controller, ControllerListener

Constructor Index

o **StopByRequestEvent**(Controller, int, int, int, Time)

Constructors

o **StopByRequestEvent**

```
public StopByRequestEvent(Controller from,
                          int previous,
                          int current,
                          int target,
                          Time mediaTime)
```

Class javax.media.StopEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
           |
           +----javax.media.TransitionEvent
                   |
                   +----javax.media.StopEvent
```

---

public class **StopEvent**
extends TransitionEvent

`StopEvent` is a `ControllerEvent` that indicates that a `Controller` has stopped.

Version:
1.28, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **StopEvent**(Controller, int, int, int, Time)

Method Index

o **getMediaTime**()
Get the clock time (*media time*) that was passed into the constructor.

Constructors

o **StopEvent**

```
public StopEvent(Controller from,
                 int previous,
                 int current,
                 int target,
                 Time mediaTime)
```

Parameters:
from – The `Controller` that generated this event.
mediaTime – The *media time* at which the `Controller` stopped.

Methods

o **getMediaTime**

```
public Time getMediaTime()
```

Get the clock time (*media time*) that was passed into the constructor.

Returns:
The *mediaTime* at which the `Controller` stopped.

Class javax.media.StopTimeChangeEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.StopTimeChangeEvent
```

---

public class **StopTimeChangeEvent**
extends ControllerEvent

A `StopTimeChangeEvent` is generated by a `Controller` when its stop time has changed.

Version:
1.12, 97/08/25.
See Also:
Controller, ControllerListener

Constructor Index

o **StopTimeChangeEvent**(Controller, Time)

Method Index

o **getStopTime**()
Get the new stop−time for the `Controller` that generated this event.

Constructors

o **StopTimeChangeEvent**

```
public StopTimeChangeEvent(Controller from,
                           Time newStopTime)
```

Methods

o **getStopTime**

```
public Time getStopTime()
```

Get the new stop−time for the `Controller` that generated this event.

Returns:
The new stop time for the `Controller` that generated this event.

Class javax.media.StopTimeSetError

```
java.lang.Object
   |
   +----java.lang.Throwable
           |
           +----java.lang.Error
                   |
                   +----javax.media.MediaError
                           |
                           +----javax.media.StopTimeSetError
```

---

public class **StopTimeSetError**
extends MediaError

`StopTimeSetError` is thrown when the stop time has been set on a *Started* `Clock` and `setStopTime` is invoked again.

Version:
1.10, 97/08/23.

Constructor Index

o **StopTimeSetError**(String)

Constructors

o **StopTimeSetError**

```
public StopTimeSetError(String reason)
```

Class javax.media.Time

```
java.lang.Object
   |
   +----javax.media.Time
```

---

public class **Time**
extends Object

`Time` abstracts time in the Java Media framework.

Version:
1.10, 97/08/28.
See Also:
Clock, TimeBase

Variable Index

o **nanoseconds**
Time is kept to a granularity of nanoseconds.
o **ONE_SECOND**

Constructor Index

o **Time**(double)
Construct a time in seconds.
o **Time**(long)
Construct a time in nanoseconds.

Method Index

o **getNanoseconds**()
Get the time value in nanoseconds.
o **getSeconds**()
Get the time value in seconds.
o **secondsToNanoseconds**(double)
Convert seconds to nanoseconds.

Variables

o **ONE_SECOND**

```
public static final long ONE_SECOND
```

o **nanoseconds**

```
protected long nanoseconds
```

Time is kept to a granularity of nanoseconds. Converions to and from this value are done to implement construction or query in seconds.

Constructors

o **Time**

```
public Time(long nano)
```

Construct a time in nanoseconds.

Parameters:
nano – Number of nanoseconds for this time.

o **Time**

```
public Time(double seconds)
```

Construct a time in seconds.

Parameters:
seconds – Time specified in seconds.

Methods

o **secondsToNanoseconds**

```
protected long secondsToNanoseconds(double seconds)
```

Convert seconds to nanoseconds.

o **getNanoseconds**

```
public long getNanoseconds()
```

Get the time value in nanoseconds.

Returns:
The time in nanoseconds.

o **getSeconds**

```
public double getSeconds()
```

Get the time value in seconds.

Interface javax.media.TimeBase

public interface **TimeBase**

A `TimeBase` is a constantly ticking source of time, much like a crystal.

Unlike a `Clock`, a `TimeBase` cannot be temporally transformed, reset, or stopped.

Version:
1.13, 97/08/25.
See Also:
Clock

Method Index

o **getNanoseconds**()
Get the current time of the `TimeBase` specified in nanoseconds.
o **getTime**()
Get the current time of this `TimeBase`.

Methods

o **getTime**

```
public abstract Time getTime()
```

Get the current time of this `TimeBase`.

Returns:
the current `TimeBase` time.

o **getNanoseconds**

```
public abstract long getNanoseconds()
```

Get the current time of the `TimeBase` specified in nanoseconds.

Returns:
the current `TimeBase` time in nanoseocnds.

Class javax.media.TransitionEvent

```
java.lang.Object
   |
   +----javax.media.ControllerEvent
            |
            +----javax.media.TransitionEvent
```

public class **TransitionEvent**
extends ControllerEvent

`TransitionEvent` is a `ControllerEvent` that indicates that a `Controller` has changed state.

Version:
1.10, 97/08/23
See Also:
Controller, ControllerListener

Constructor Index

o **TransitionEvent**(Controller, int, int, int)
Construct a new `TransitionEvent`.

Method Index

o **getCurrentState**()
Get the `Controller's` state at the time this event was generated
o **getPreviousState**()
Get the state that the `Controller` was in before this event occurred.
o **getTargetState**()
Get the `Controller's` target state at the time this event was generated.

Constructors

o **TransitionEvent**

```
public TransitionEvent(Controller from,
                       int previous,
                       int current,
                       int target)
```

Construct a new `TransitionEvent`.

Parameters:
from – The `Controller` that is generating this event.
previous – The state that the `Controller` was in before this event.
current – The state that the `Controller` is in as a result of this event.
target – The state that the `Controller` is heading to.

Methods

o **getPreviousState**

```
public int getPreviousState()
```

Get the state that the `Controller` was in before this event occurred.

Returns:
The `Controller's` previous state.

o **getCurrentState**

```
public int getCurrentState()
```

Get the `Controller's` state at the time this event was generated

Returns:
The `Controller's` current state.

o **getTargetState**

```
public int getTargetState()
```

Get the `Controller's` target state at the time this event was generated.

Returns:
The `Controller's` target state.

Class javax.media.protocol.ContentDescriptor

```
java.lang.Object
   |
   +----javax.media.protocol.ContentDescriptor
```

---

public class **ContentDescriptor**
extends Object

A `ContentDescriptor` identifies media data containers.

Version:
1.10, 97/08/26.
See Also:
SourceStream

Variable Index

o **CONTENT_UNKNOWN**
o **typeName**

Constructor Index

o **ContentDescriptor**(String)
Create a content descriptor with the specified name.

Method Index

o **getContentType**()
Obtain a string that represents the content−name for this descriptor.
o **mimeTypeToPackageName**(String)
Map a MIME content−type to an equivalent string of class−name components.

Variables

o **CONTENT_UNKNOWN**

```
public static final String CONTENT_UNKNOWN
```

o **typeName**

```
protected String typeName
```

Constructors

o **ContentDescriptor**

```
public ContentDescriptor(String cdName)
```

Create a content descriptor with the specified name.

To create a `ContentDescriptor` from a MIME type, use the `mimeTypeToPackageName` static member.

Parameters:
cdName − The name of the content−type.

Methods

o **getContentType**

```
public String getContentType()
```

Obtain a string that represents the content−name for this descriptor.

Returns:
The content−type name.

o **mimeTypeToPackageName**

```
protected static final String mimeTypeToPackageName(String mimeType)
```

Map a MIME content−type to an equivalent string of class−name components.

The MIME type is mapped to a string by:
1. Replacing all slashes with a period.
2. Converting all alphabetic characters to lower case.
3. Converting all non−alpha−numeric characters other than periods to underscores (_).

For example, "text/html" would be converted to "text.html"

Parameters:

mimeType – The MIME type to map to a string.

Interface javax.media.protocol.Controls

public interface **Controls**

`Controls` provides an interface for obtaining objects by interface or class name. This is useful in the case where support for a particular interface cannot be determined at runtime, or where a different object is required to implement the behavior. The `object` returned from `getControl` is assumed to control the `object` that `getControl` was invoked on.

Version:
1.4, 97/08/28.

Method Index

o **getControl**(String)
Obtain the object that implements the specified `Class` or `Interface` The full class or interface name must be used.
o **getControls**()
Obtain the collection of objects that control the object that implements this interface.

Methods

o **getControls**

```
public abstract Object[] getControls()
```

Obtain the collection of objects that control the object that implements this interface.

If no controls are supported, a zero length array is returned.

Returns:
the collection of object controls

o **getControl**

```
public abstract Object getControl(String controlType)
```

Obtain the object that implements the specified `Class` or `Interface` The full class or interface name must be used.

If the control is not supported then `null` is returned.

Returns:
the object that implements the control, or `null`.

Class javax.media.protocol.DataSource

```
java.lang.Object
   |
   +----javax.media.protocol.DataSource
```

---

public abstract class **DataSource**
extends Object
implements Controls, Duration

A `DataSource` is an abstraction for media protocol−handlers. `DataSource` manages the life−cycle of the media source by providing a simple connection protocol.

Source Controls

A `DataSource` might support an operation that is not part of the `DataSource` class definition. For example a source could support positioning its media to a particular time. Some operations are dependent on the data stream that the source is managing, and support cannot be determined until after the source has been connected.

To obtain all of the objects that provide control over a `DataSource`, use `getControls` which returns an array of `Object` To determine if a particular kind of control is available and obtain the object that implements it, use `getControl` which takes the name of the Class or Interface that of the desired control.

Version:
1.16, 97/08/26
See Also:
Manager, DefaultPlayerFactory, Positionable, RateConfigureable

---

Constructor Index

o **DataSource**()
A no−argument constructor required by pre 1.1 implementations so that this class can be instantiated by calling `Class.newInstance`.
o **DataSource**(MediaLocator)
Construct a `DataSource` from a `MediaLocator`.

Method Index

o **connect**()
Open a connection to the source described by the `MediaLocator`.
o **disconnect**()
Close the connection to the source described by the locator.
o **getContentType**()
Get a string that describes the content−type of the media that the source is providing.
o **getControl**(String)
Obtain the object that implements the specified `Class` or `Interface` The full class or interface name must be used.
o **getControls**()
Obtain the collection of objects that control the object that implements this interface.
o **getDuration**()
Get the duration of the media represented by this object.
o **getLocator**()
Get the `MediaLocator` that describes this source.
o **initCheck**()
Check to see if this connection has been initialized with a `MediaLocator`.
o **setLocator**(MediaLocator)
Set the connection `source` for this `DataSource`.
o **start**()
Initiate data−transfer.
o **stop**()
Stop the data−transfer.

Constructors

o **DataSource**

```
public DataSource()
```

A no−argument constructor required by pre 1.1 implementations so that this class can be instantiated by calling `Class.newInstance`.

o **DataSource**

```
public DataSource(MediaLocator source)
```

Construct a `DataSource` from a `MediaLocator`. This method should be overloaded by subclasses; the default implementation just keeps track of the `MediaLocator`.

Parameters:
source − The `MediaLocator` that describes the `DataSource`.

Methods

o **setLocator**

```
public void setLocator(MediaLocator source)
```

Set the connection `source` for this `DataSource`. This method should only be called once; an error is thrown if the locator has already been set.

Parameters:
source – The `MediaLocator` that describes the media source.

o **getLocator**

```
public MediaLocator getLocator()
```

Get the `MediaLocator` that describes this source. Returns `null` if the locator hasn't been set. (Very unlikely.)

Returns:
The `MediaLocator` for this source.

o **initCheck**

```
protected void initCheck()
```

Check to see if this connection has been initialized with a `MediaLocator`. If the connection hasn't been initialized, `initCheck` throws an `UninitializedError`. Most methods should call `initCheck` on entry.

o **getContentType**

```
public abstract String getContentType()
```

Get a string that describes the content–type of the media that the source is providing.

It is an error to call `getContentType` if the source is not connected.

Returns:
The name that describes the media content.

o **connect**

```
public abstract void connect() throws IOException
```

Open a connection to the source described by the `MediaLocator`.

The `connect` method initiates communication with the source.

Throws: IOException
Thrown if there are IO problems when `connect` is called.

o **disconnect**

```
public abstract void disconnect()
```

Close the connection to the source described by the locator.

The `disconnect` method frees resources used to maintain a connection to the source. If no resources are in use, `disconnect` is ignored. If `stop` hasn't already been called, calling `disconnect` implies a stop.

o **start**

```
public abstract void start() throws IOException
```

Initiate data–transfer. The `start` method must be called before data is available. (You must call `connect` before calling `start`.)

Throws: IOException
Thrown if there are IO problems with the source when `start` is called.

o **stop**

```
public abstract void stop() throws IOException
```

Stop the data–transfer. If the source has not been connected and started, `stop` does nothing.

Interface javax.media.protocol.Positionable

public interface **Positionable**

A `DataSource` implements the `Positionable` interface if it supports changing the media position within the stream.

Version:
1.6, 97/08/23.
See Also:
Datasource

Variable Index

o **RoundDown**
o **RoundNearest**
o **RoundUp**

Method Index

o **isRandomAccess**()
Find out if this source can be repositioned to any point in the stream.
o **setPosition**(Time, int)
Set the position to the specified time.

Variables

o **RoundUp**

```
public static final int RoundUp
```

o **RoundDown**

```
public static final int RoundDown
```

o **RoundNearest**

```
public static final int RoundNearest
```

Methods

o **setPosition**

```
public abstract Time setPosition(Time where,
                                 int rounding)
```

Set the position to the specified time. Returns the rounded position that was actually set.

Parameters:
time – The new position in the stream.
round – The rounding technique to be used: RoundUp, RoundDown, RoundNearest.
Returns:
The actual position set.

o **isRandomAccess**

```
public abstract boolean isRandomAccess()
```

Find out if this source can be repositioned to any point in the stream. If not, the source can only be repositioned to the beginning of the stream.

Returns:
Returns `true` if the source is random access; `false` if the source can only be reset to the beginning of the stream.

Class javax.media.protocol.PullDataSource

```
java.lang.Object
   |
   +----javax.media.protocol.DataSource
           |
           +----javax.media.protocol.PullDataSource
```

public abstract class **PullDataSource**
extends DataSource

Abstracts a media data−source that only supports pull data−streams.

Version:
1.5, 97/08/23.
See Also:
Manager, Player, DefaultPlayerFactory, DataSource

Constructor Index

o **PullDataSource**()

Method Index

o **getStreams**()
Get the collection of streams that this source manages.

Constructors

o **PullDataSource**

```
public PullDataSource()
```

Methods

o **getStreams**

```
public abstract PullSourceStream[] getStreams()
```

Get the collection of streams that this source manages. The collection of streams is entirely content dependent. The MIME type of this `DataSource` provides the only indication of what streams can be available on this connection.

Returns:
The collection of streams for this source.

Interface javax.media.protocol.PullSourceStream

public interface **PullSourceStream**
extends SourceStream

Abstracts a read interface that data is pulled from.

Version:
1.8, 97/08/23.
See Also:
PullDataSource

Method Index

o **read**(byte[], int, int)
Block and read data from the stream.
o **willReadBlock**()
Find out if data is available now.

Methods

o **willReadBlock**

```
public abstract boolean willReadBlock()
```

Find out if data is available now. Returns `true` if a call to `read` would block for data.

Returns:
Returns `true` if read would block; otherwise returns `false`.

o **read**

```
public abstract int read(byte buffer[],
                         int offset,
                         int length) throws IOException
```

Block and read data from the stream.

Reads up to `length` bytes from the input stream into an array of bytes. If the first argument is `null`, up to `length` bytes are read and discarded. Returns −1 when the end of the media is reached. This method only returns 0 if it was called with a `length` of 0.

Parameters:
buffer − The buffer to read bytes into.
offset − The offset into the buffer at which to begin writing data.
length − The number of bytes to read.
Returns:
The number of bytes read, −1 indicating the end of stream, or 0 indicating `read` was called with `length` 0.

Class javax.media.protocol.PushDataSource

```
java.lang.Object
   |
   +----javax.media.protocol.DataSource
           |
           +----javax.media.protocol.PushDataSource
```

public abstract class **PushDataSource**
extends DataSource

Abstracts a data source that manages `PushDataStreams`.

Version:
1.5, 97/08/23.
See Also:
Manager, Player, DefaultPlayerFactory, DataSource

Constructor Index

o **PushDataSource**()

Method Index

o **getStreams**()
Get the collection of streams that this source manages.

Constructors

o **PushDataSource**

```
public PushDataSource()
```

Methods

o **getStreams**

```
public abstract PushSourceStream[] getStreams()
```

Get the collection of streams that this source manages. The collection of streams is entirely content dependent. The `ContentDescriptor` of this `DataSource` provides the only indication of what streams can be available on this connection.

Returns:
The collection of streams for this source.

Interface javax.media.protocol.PushSourceStream

public interface **PushSourceStream**
extends SourceStream

Abstracts a read interface that pushes data.

Version:
1.7, 97/08/25.
See Also:
PushDataSource

Method Index

o **getMinimumTransferSize**()
Determine the size of the buffer needed for the data transfer.
o **read**(byte[], int, int)
Read from the stream without blocking.
o **setTransferHandler**(SourceTransferHandler)
Register an object to service data transfers to this stream.

Methods

o **read**

```
public abstract int read(byte buffer[],
                         int offset,
                         int length)
```

Read from the stream without blocking. Returns −1 when the end of the media is reached.

Parameters:
buffer − The buffer to read bytes into.
offset − The offset into the buffer at which to begin writing data.
length − The number of bytes to read.
Returns:
The number of bytes read or −1 when the end of stream is reached.

o **getMinimumTransferSize**

```
public abstract int getMinimumTransferSize()
```

Determine the size of the buffer needed for the data transfer. This method is provided so that a transfer handler can determine how much data, at a minimum, will be available to transfer from the source. Overflow and data loss is likely to occur if this much data isn't read at transfer time.

Returns:
The size of the data transfer.

o **setTransferHandler**

```
public abstract void setTransferHandler(SourceTransferHandler transferHandler)
```

Register an object to service data transfers to this stream.

If a handler is already registered when `setTransferHandler` is called, the handler is replaced; there can only be one handler at a time.

Parameters:
transferHandler − The handler to transfer data to.

Interface javax.media.protocol.RateConfiguration

public interface **RateConfiguration**

A configuration of streams for a particular rate.

Version:
1.7, 97/08/28.
See Also:
DataSource, RateConfigureable

Method Index

o **getRate**()
Get the `RateRange` for this configuration.
o **getStreams**()
Get the streams that will have content at this rate.

Methods

o **getRate**

```
public abstract RateRange getRate()
```

Get the `RateRange` for this configuration.

Returns:
The rate supported by this configuration.

o **getStreams**

```
public abstract SourceStream[] getStreams()
```

Get the streams that will have content at this rate.

Returns:
The streams supported at this rate.

All Packages Class Hierarchy This Package Previous Next Index

Interface javax.media.protocol.RateConfigureable

public interface **RateConfigureable**

`DataSources` support the `RateConfigureable` interface if they use different rate−configurations to support multiple media display speeds.

Version:
1.7, 97/08/26.
See Also:
DataSource, RateConfiguration, RateRange

Method Index

o **getRateConfigurations**()
Get the rate configurations that this object supports.
o **setRateConfiguration**(RateConfiguration)
Set a new `RateConfiguration`.

Methods

o **getRateConfigurations**

```
public abstract RateConfiguration[] getRateConfigurations()
```

Get the rate configurations that this object supports. There must always be one and only one for a `RateConfiguration` that covers a rate of 1.0.

Returns:
The collection of `RateConfigurations` that this source supports.

o **setRateConfiguration**

```
public abstract RateConfiguration setRateConfiguration(RateConfiguration config)
```

Set a new `RateConfiguration`. The new configuration should have been obtained by calling `getRateConfigurations`. Returns the actual `RateConfiguration` used.

Parameters:
config – The `RateConfiguration` to use.

Returns:
The actual `RateConfiguration` used by the source.

Class javax.media.protocol.RateRange

```
java.lang.Object
   |
   +----javax.media.protocol.RateRange
```

---

public class **RateRange**
extends Object

Describes the speed at which data flows.

Version:
1.6, 97/08/23.

Constructor Index

o **RateRange**(float, float, float, boolean)
Constructor using required values.
o **RateRange**(RateRange)
Copy constructor.

Method Index

o **getCurrentRate**()
Get the current rate.
o **getMaximumRate**()
Get the maximum rate supported by this range.
o **getMinimumRate**()
Get the minimum rate supported by this range.
o **isExact**()
Determine whether or not the source will maintain a constant speed when using this rate.
o **setCurrentRate**(float)
Set the current rate.

Constructors

o **RateRange**

```
public RateRange(RateRange r)
```

Copy constructor.

o **RateRange**

```
public RateRange(float init,
                 float min,
                 float max,
                 boolean isExact)
```

Constructor using required values.

Parameters:
init – The initial value for this rate.
min – The minimum value that this rate can take.
max – The maximum value that this rate can take.
isExact – Set to `true` if the source rate does not vary when using this rate range.

Methods

o **setCurrentRate**

```
public float setCurrentRate(float rate)
```

Set the current rate. Returns the rate that was actually set. This implementation just returns the specified rate, subclasses should return the rate that was actually set.

Parameters:
rate – The new rate.

o **getCurrentRate**

```
public float getCurrentRate()
```

Get the current rate.

Returns:
The current rate.

o **getMinimumRate**

```
public float getMinimumRate()
```

Get the minimum rate supported by this range.

Returns:
The minimum rate.

o **getMaximumRate**

```
public float getMaximumRate()
```

Get the maximum rate supported by this range.

Returns:
The maximum rate.

o **isExact**

```
public boolean isExact()
```

Determine whether or not the source will maintain a constant speed when using this rate. If the rate varies, synchronization is usually impractical.

Returns:
Returns `true` if the source will maintain a constant speed at this rate.

Interface javax.media.protocol.Seekable

public interface **Seekable**

A `SourceStream` will implement this interface if it is capable of seeking to a particular position in the stream.

Version:
1.6, 97/08/23.
See Also:
SourceStream

Method Index

o **isRandomAccess**()
Find out if this source can position anywhere in the stream.
o **seek**(long)
Seek to the specified point in the stream.
o **tell**()
Obtain the current point in the stream.

Methods

o **seek**

```
public abstract long seek(long where)
```

Seek to the specified point in the stream.

Parameters:
where – The position to seek to.
Returns:
The new stream position.

o **tell**

```
public abstract long tell()
```

Obtain the current point in the stream.

o **isRandomAccess**

```
public abstract boolean isRandomAccess()
```

Find out if this source can position anywhere in the stream. If the stream is not random access, it can only be repositioned to the beginning.

Returns:
Returns `true` if the stream is random access, `false` if the stream can only be reset to the beginning.

Interface javax.media.protocol.SourceStream

public interface **SourceStream**
extends Controls

Abstracts a single stream of media data.

Stream Controls

A `SourceStream` might support an operation that is not part of the `SourceStream` definition. For example a stream might support seeking to a particular byte in the stream. Some operations are dependent on the stream data, and support cannot be determined until the stream is in use.

To obtain all of the objects that provide control over a stream use `getControls`. To determine if a particular kind of control is available, and obtain the object that implements the control use `getControl`.

Version:
1.12, 97/08/28.
See Also:
DataSource, PushSourceStream, PullSourceStream, Seekable

Variable Index

o **LENGTH_UNKNOWN**

Method Index

o **endOfStream**()
Find out if the end of the stream has been reached.
o **getContentDescriptor**()
Get the current content type for this stream.
o **getContentLength**()
Get the size, in bytes, of the content on this stream.

Variables

o **LENGTH_UNKNOWN**

```
public static final long LENGTH_UNKNOWN
```

Methods

o **getContentDescriptor**

```
public abstract ContentDescriptor getContentDescriptor()
```

Get the current content type for this stream.

Returns:
The current `ContentDescriptor` for this stream.

o **getContentLength**

```
public abstract long getContentLength()
```

Get the size, in bytes, of the content on this stream. LENGTH_UNKNOWN is returned if the length is not known.

Returns:
The content length in bytes.

o **endOfStream**

```
public abstract boolean endOfStream()
```

Find out if the end of the stream has been reached.

Returns:
Returns `true` if there is no more data.

Interface javax.media.protocol.SourceTransferHandler

public interface **SourceTransferHandler**

Implements the callback from a `PushSourceStream`.

Version:
1.5, 97/08/23.
See Also:
PushSourceStream

Method Index

o **transferData**(PushSourceStream)
Transfer new data from a `PushSourceStream`.

Methods

o **transferData**

```
public abstract void transferData(PushSourceStream stream)
```

Transfer new data from a `PushSourceStream`.

Parameters:
stream – The stream that is providing the data.

Class javax.media.protocol.URLDataSource

```
java.lang.Object
   |
   +----javax.media.protocol.DataSource
           |
           +----javax.media.protocol.PullDataSource
                   |
                   +----javax.media.protocol.URLDataSource
```

---

public class **URLDataSource**
extends PullDataSource

A default data−source created directly from a `URL` using `URLConnection`.

Version:
1.19, 97/08/28.
See Also:
URL, URLConnection, InputSourceStream

Variable Index

o **conn**
o **connected**
o **contentType**
o **sources**

Constructor Index

o **URLDataSource**()
Implemented by subclasses.
o **URLDataSource**(URL)
Construct a `URLDataSource` directly from a `URL`.

Method Index

o **connect**()
Initialize the connection with the source.
o **disconnect**()
Disconnect the source.
o **getContentType**()
Return the content type name.
o **getControl**(String)
Returns null, because this source doesn't provide any controls.
o **getControls**()
Returns an empty array, because this source doesn't provide any controls.
o **getDuration**()
Returns `Duration.DURATION_UNKNOWN`.
o **getStreams**()
Get the collection of streams that this source manages.
o **start**()
Initiate data−transfer.
o **stop**()
Stops the

Variables

o **conn**

```
protected URLConnection conn
```

o **contentType**

```
protected ContentDescriptor contentType
```

o **sources**

```
protected URLSourceStream sources[]
```

o **connected**

```
protected boolean connected
```

Constructors

o **URLDataSource**

```
protected URLDataSource()
```

Implemented by subclasses.

o **URLDataSource**

```
public URLDataSource(URL url) throws IOException
```

Construct a `URLDataSource` directly from a `URL`.

Methods

o **getStreams**

```
public PullSourceStream[] getStreams()
```

Get the collection of streams that this source manages.

Overrides:
getStreams in class PullDataSource

o **connect**

```
public void connect() throws IOException
```

Initialize the connection with the source.

Throws: IOException
Thrown if there are problems setting up the connection.
Overrides:
connect in class DataSource

o **getContentType**

```
public String getContentType()
```

Return the content type name.

Returns:
The content type name.
Overrides:
getContentType in class DataSource

o **disconnect**

```
public void disconnect()
```

Disconnect the source.

Overrides:
disconnect in class DataSource

o **start**

```
public void start() throws IOException
```

Initiate data−transfer.

Overrides:
start in class DataSource

o **stop**

```
public void stop() throws IOException
```

Stops the

Overrides:
stop in class DataSource

o **getDuration**

```
public Time getDuration()
```

Returns `Duration.DURATION_UNKNOWN`. The duration is not available from an `InputStream`.

Returns:
`Duration.DURATION_UNKNOWN`.
Overrides:
getDuration in class DataSource

o **getControls**

```
public Object[] getControls()
```

Returns an empty array, because this source doesn't provide any controls.

Returns:
empty `Object` array.
Overrides:
getControls in class DataSource

o **getControl**

```
public Object getControl(String controlName)
```

Returns null, because this source doesn't provide any controls.

Overrides:
getControl in class DataSource