Cisco J TAPI Developer Guide for Cisco CallManager 3.3(2)

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
    800 553-NETS (6387)
Fax: 408 526-4100

Text Part Number: OL-3022-01
CHAPTER 1

Overview 1-1

Cisco J TAPI 1-2
CiscoObjectContainer 1-2
J tapiPeer and Provider 1-3
    Initialization 1-4
    Shutdown 1-4
Provider.getTerminals() 1-4
Provider.getAddresses() 1-5
Changes to the User Control List in the Directory 1-5

Addresses and Terminals 1-5
    Address and Terminal Relationship 1-6
    Unobserved Addresses and Terminals 1-7

Connection 1-7
TerminalConnection 1-8
CiscoConnectionID 1-8
CiscoCallID 1-8
Threaded CallBacks 1-8
    Querying Dynamic Objects 1-9
        callChangeEvent() 1-9
    CiscoConsultCall 1-9
    CiscoTransferStartEv 1-10

Alarm Services 1-10
Application Control of J TAPI Parameters 1-10
    J tapi.ini Parameters 1-11
        Dynamic Trace Enabling Using Jtprefs 1-13

Call Forward Setting 1-13
Call Park 1-13
    Park Retrieval 1-14
Contents

Park Reminder 1-14
Park DN Monitor 1-14
CiscoJtapiExceptions 1-15
Cisco J TAPI Install Internationalization 1-15
Clear Calls Interface 1-15
Device Recovery 1-15
  Device Recovery for phones 1-16
CTI RoutePoints 1-16
CTI Ports 1-16
Directory Change Notification 1-17
Enable or Disable Ringer 1-17
Transfer and Conference Extensions 1-17
  Transfer 1-17
    CiscoTransferStartEv 1-18
    CiscoTransferEndEv 1-18
  Transfer Scenarios 1-18
Conference 1-22
  Cisco Extensions 1-23
  Conference Scenarios 1-24
  Conference Events 1-25
  Transfer and Conference Enhancement 1-26
Consult Without Media 1-26
Media Termination Extensions 1-27
Cisco CallManager Media Endpoint Model 1-28
  Payload and Parameter Negotiation 1-28
Initialization 1-29
  Payload Selection 1-29
  Receive Channel Allocation 1-30
Starting Transmission and Reception 1-30
Contents

Stopping Transmission and Reception 1-31
CiscoMediaTerminal 1-31
Provisioning 1-31
Registration 1-32
Adding Observers 1-34
Accepting Calls 1-35
Receiving and Responding to Media Flow Events 1-35
Inbound Call Media Flow Event Diagram 1-36
Cisco IP Telephony Solutions RTP Implementation 1-37
Redirect 1-37
Routing 1-38
Cisco Route Session Implementation 1-39
Select Route Timer 1-40
Forwarding Timer 1-40
RouteSession Extension 1-41
Caller Options Summary 1-41
Fault Tolerance When Using Route Points 1-42
Redundancy 1-42
Cluster Abstraction 1-42
Cisco CallManager Server Failure 1-44
Redundancy in CTIManagers 1-46
Invoking CTIManager Redundancy 1-47
CTIManager Failure 1-48
Heartbeats 1-49
DisplayName Interface 1-49
SetMessageWaiting Interface 1-50
Quite Clear 1-51
GetCallInfo Interface on Address 1-51
DeleteCall Interface 1-52
GetGlobalCallID 1-52
GetCallID in RTP Events 1-53
XSI Object Pass Through 1-53

Cisco J TAPI Implementation 2-1
Cisco J TAPI Extensions Hierarchy 2-1
Class Hierarchy 2-1
Interface Hierarchy 2-2
Class com.cisco.jtapi.extensions 2-8
CiscoAddrCreatedEv 2-8
Declaration 2-8
All Superinterfaces 2-8
Description 2-8
Fields 2-9
Methods 2-9
CiscoAddress 2-10
Declaration 2-10
All Superinterfaces 2-10
Description 2-10
Fields 2-12
Methods 2-13
CiscoAddressCallInfo 2-15
Declaration 2-15
Constructors 2-16
Methods 2-17
CiscoAddressObserver 2-17
Declaration 2-17
All Superinterfaces 2-17
Description 2-18
CiscoAddrEv 2-18
  Declaration 2-18
  All Superinterfaces 2-18
  All Known Subinterfaces 2-18
  Description 2-19
CiscoAddrInServiceEv 2-19
  Declaration 2-19
  All Superinterfaces 2-20
  Description 2-20
  Fields 2-21
CiscoAddrOutOfServiceEv 2-21
  Declaration 2-21
  All Superinterfaces 2-21
  Description 2-21
  Fields 2-22
CiscoAddrRemovedEv 2-22
  Declaration 2-22
  All Superinterfaces 2-23
  Description 2-23
  Fields 2-24
  Methods 2-24
CiscoCall 2-24
  Declaration 2-24
  All Superinterfaces 2-24
  All Known Subinterfaces 2-25
  Description 2-25
  Methods 2-27
CiscoCallEv 2-28
  Declaration 2-28
  All Superinterfaces 2-28
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Known Subinterfaces</td>
<td>2-28</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>2-28</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td>2-31</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>2-37</td>
<td></td>
</tr>
<tr>
<td>CiscoCallID</td>
<td>2-37</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>2-37</td>
<td></td>
</tr>
<tr>
<td>All Superinterfaces</td>
<td>2-37</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>2-37</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>2-38</td>
<td></td>
</tr>
<tr>
<td>CiscoConferenceEndEv</td>
<td>2-39</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>2-39</td>
<td></td>
</tr>
<tr>
<td>All Superinterfaces</td>
<td>2-39</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>2-39</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td>2-42</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>2-42</td>
<td></td>
</tr>
<tr>
<td>CiscoConferenceStartEv</td>
<td>2-43</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>2-43</td>
<td></td>
</tr>
<tr>
<td>All Superinterfaces</td>
<td>2-43</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>2-43</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td>2-46</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>2-46</td>
<td></td>
</tr>
<tr>
<td>CiscoConnection</td>
<td>2-47</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>2-47</td>
<td></td>
</tr>
<tr>
<td>All Superinterfaces</td>
<td>2-47</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>2-47</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td>2-50</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>2-52</td>
<td></td>
</tr>
<tr>
<td>CiscoConnectionID</td>
<td>2-56</td>
<td></td>
</tr>
<tr>
<td>Declaration</td>
<td>2-56</td>
<td></td>
</tr>
<tr>
<td>All Superinterfaces</td>
<td>2-56</td>
<td></td>
</tr>
</tbody>
</table>
Contents

Description 2-56
Methods 2-57
CiscoConsultCall 2-57
Declaration 2-57
All Superinterfaces 2-58
Description 2-58
Methods 2-59
CiscoConsultCallActiveEv 2-61
Declaration 2-61
All Superinterfaces 2-61
Description 2-61
Fields 2-63
Methods 2-63
CiscoEv 2-64
Declaration 2-64
All Superinterfaces 2-64
All Known Subinterfaces 2-64
Description 2-65
CiscoG711MediaCapability 2-65
Declaration 2-65
Description 2-66
Fields 2-67
Constructors 2-67
CiscoG723MediaCapability 2-68
Declaration 2-68
Description 2-68
Fields 2-69
Constructors 2-70
Methods 2-70
Contents

CiscoG729MediaCapability 2-71
Declaration 2-71
Description 2-71
Fields 2-72
Constructors 2-73
CiscoGSM MediaCapability 2-73
Declaration 2-73
Description 2-73
Fields 2-75
Constructors 2-75
CiscoJtapiException 2-75
Declaration 2-75
Description 2-75
Fields 2-79
Methods 2-86
CiscoJtapiPeer 2-87
Declaration 2-87
All Superinterfaces 2-88
Description 2-88
Methods 2-89
CiscoJtapiProperties 2-89
Declaration 2-89
Description 2-89
Methods 2-93
CiscoJtapiVersion 2-101
Declaration 2-101
Description 2-101
Constructors 2-102
Methods 2-102
CiscoMediaCapability
  Declaration  2-103
  Direct Known Subclasses  2-103
  Description  2-104
  Fields  2-105
  Constructors  2-106
  Methods  2-106

CiscoMediaTerminal
  Declaration  2-107
  All Superinterfaces  2-107
  Description  2-107
  Methods  2-109

CiscoObjectContainer
  Declaration  2-111
  All Known Subinterfaces  2-111
  Description  2-111
  Methods  2-112

CiscoOutOfServiceEv
  Declaration  2-112
  All Superinterfaces  2-112
  All Known Subinterfaces  2-113
  Description  2-113
  Fields  2-114

CiscoProvCallParkEv
  Declaration  2-114
  All Superinterfaces  2-115
  Description  2-115
  Fields  2-116
  Methods  2-117

CiscoProvEv
  2-118
Declaration 2-118
All Superinterfaces 2-118
All Known Subinterfaces 2-119
Description 2-119

CiscoProvFeatureEv 2-120
Declaration 2-120
All Superinterfaces 2-120
All Known Subinterfaces 2-120
Methods 2-121

CiscoProvFeatureID 2-121
Declaration 2-121
Fields 2-121

CiscoProvFeatureUnRegisteredEv 2-122
Declaration 2-122
All Superinterfaces 2-122
Description 2-122
Fields 2-123
Methods 2-123

CiscoProvider 2-123
Declaration 2-123
All Superinterfaces 2-124
Description 2-124
Methods 2-125

CiscoProviderCapabilities 2-129
Declaration 2-129
All Superinterfaces 2-129
Description 2-129

CiscoProviderObserver 2-130
Declaration 2-130
Contents

Fields 2-143
Methods 2-143
CiscoRTPInputStoppedEv 2-143
Declaration 2-143
All Superinterfaces 2-143
Description 2-143
Fields 2-144
Methods 2-145
CiscoRTPOutputProperties 2-145
Declaration 2-145
Description 2-145
Methods 2-146
CiscoRTPOutputStartedEv 2-148
Declaration 2-148
All Superinterfaces 2-148
Description 2-148
Fields 2-149
Methods 2-149
CiscoRTPOutputStoppedEv 2-150
Declaration 2-150
All Superinterfaces 2-150
Description 2-150
Fields 2-151
Methods 2-151
CiscoRTPParams 2-151
Declaration 2-151
Constructors 2-152
Methods 2-152
CiscoRTPPayload 2-153
<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>2-153</td>
<td>Description</td>
<td>2-153</td>
</tr>
<tr>
<td>Declaration</td>
<td>2-157</td>
<td>Description</td>
<td>2-157</td>
</tr>
<tr>
<td>CiscoSynchronousObserver</td>
<td>2-157</td>
<td>Declaration</td>
<td>2-157</td>
</tr>
<tr>
<td>CiscoTermCreatedEv</td>
<td>2-158</td>
<td>All Superinterfaces</td>
<td>2-159</td>
</tr>
<tr>
<td>CiscoTermDataEv</td>
<td>2-160</td>
<td>All Superinterfaces</td>
<td>2-160</td>
</tr>
<tr>
<td>CiscoTermEv</td>
<td>2-162</td>
<td>All Known Subinterfaces</td>
<td>2-162</td>
</tr>
<tr>
<td>CiscoTerminal</td>
<td>2-163</td>
<td>Description</td>
<td>2-163</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All Superinterfaces</td>
<td>2-163</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All Known Subinterfaces</td>
<td>2-164</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fields</td>
<td>2-165</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Methods</td>
<td>2-165</td>
</tr>
</tbody>
</table>
CiscoTerminalConnection 2-168
  Declaration 2-168
  All Superinterfaces 2-168
  Description 2-168

CiscoTerminalObserver 2-169
  Declaration 2-169
  All Superinterfaces: 2-169
  Description 2-169

CiscoTermInServiceEv 2-170
  Declaration 2-170
  All Superinterfaces 2-170
  Description 2-170
  Fields 2-171

CiscoTermOutOfServiceEv 2-171
  Declaration 2-171
  All Superinterfaces 2-172
  Description 2-172
  Fields 2-173

CiscoTermRemovedEv 2-173
  Declaration 2-173
  All Superinterfaces 2-173
  Description 2-173
  Fields 2-174
  Methods 2-175

CiscoTransferEndEv 2-175
  Declaration 2-175
  All Superinterfaces 2-175
  Description 2-175
  Fields 2-177
  Methods 2-177
CiscoTransferStartEv 2-178
   Declaration 2-178
   All Superinterfaces 2-178
   Description 2-178
   Fields 2-181
   Methods 2-181

CiscoUnregistrationException 2-182
   Declaration 2-182
   All Implemented Interfaces 2-182
   Description 2-182
   Constructors 2-183

Class com.cisco.services.alarm 2-183
   Alarm 2-184
      Declaration 2-184
      All Known Implementing Classes 2-184
      Description 2-184
      Fields 2-186
      Methods 2-188
   AlarmManager 2-189
      Declaration 2-189
      Description 2-190
      Constructors 2-191
      Methods 2-192
   AlarmWriter 2-192
      Declaration 2-192
      All Known Implementing Classes 2-193
      Description 2-193
      Methods 2-193
   DefaultAlarm 2-194
Contents

Declaration 2-194
All Implemented Interfaces 2-195
Description 2-195
Constructors 2-196
Methods 2-196

DefaultAlarmWriter 2-197
Declaration 2-197
All Implemented Interfaces 2-197
Description 2-198
Constructors 2-199
Methods 2-200

ParameterList 2-201
Declaration 2-201
Description 2-202
Constructors 2-203
Methods 2-203

Class com.cisco.services.tracing 2-204

BaseTraceWriter 2-205
Declaration 2-205
All Implemented Interfaces 2-205
Direct Known Subclasses 2-206
Description 2-206
Constructors 2-207
Methods 2-208

ConditionalTrace 2-210
Declaration 2-210
All Superinterfaces 2-211
Description 2-211
Methods 2-212
Contents

ConsoleTraceWriter  2-212
  Declaration  2-212
  All Implemented Interfaces  2-212
  Description  2-213
  Constructors  2-214
  Methods  2-214

LogFileTraceWriter  2-215
  Declaration  2-215
  All Implemented Interfaces  2-215
  Description  2-215
  Fields  2-219
  Constructors  2-220
  Methods  2-221

OutputStreamTraceWriter  2-223
  Declaration  2-223
  All Implemented Interfaces  2-223
  Description  2-223
  Constructors  2-224
  Methods  2-225

SyslogTraceWriter  2-226
  Declaration  2-226
  All Implemented Interfaces  2-226
  Description  2-226
  Constructors  2-227
  Methods  2-228

Trace  2-229
  Declaration  2-229
  All Known Subinterfaces  2-229
  Description  2-229
  Fields  2-232
Contents

Methods 2-234
TraceManager 2-236
   Declaration 2-236
   Description 2-236
   Methods 2-239

TraceManagerFactory 2-241
   Declaration 2-241
   Description 2-242
   Methods 2-242

TraceModule 2-243
   Declaration 2-243
   All Known Subinterfaces 2-243
   Description 2-244
   Methods 2-244

TraceWriter 2-244
   Declaration 2-244
   All Known Subinterfaces 2-245
   All Known Implementing Classes 2-245
   Description 2-245
   Methods 2-246

TraceWriterManager 2-247
   Declaration 2-247
   All Superinterfaces 2-247
   Description 2-248
   Methods 2-249

UnconditionalTrace 2-250
   Declaration 2-250
   All Superinterfaces 2-250
   Description 2-250
## Chapter 3: JTAPI Examples

- **MakeCall.java** 3-2
- **Actor.java** 3-5
- **Originator.java** 3-10
- **Receiver.java** 3-14
- **StopSignal.java** 3-16
- **Trace.java** 3-17
- **TraceWindow.java** 3-18
- **Running makecall** 3-21
  - **Invoking makecall** 3-21
  - **On the client workstation** 3-21

### Appendix A: JTAPI Classes and Interfaces

**Cisco JTAPI Version 1.2 Classes and Interfaces**  A-2
- Core Package  A-2
- Call Center Package  A-6
- Call Center Capabilities Package  A-8
- Call Center Events Package  A-9
- Call Control Package  A-12
- Call Control Capabilities Package  A-15
- Call Control Events Package  A-16
- Capabilities Package  A-18
- Events Package  A-19
- Media Package  A-20
- Media Capabilities Package  A-21
- Media Events Package  A-22
- Unsupported Packages  A-22

**Cisco JTAPI Extension Classes and Interfaces**  A-23
- Cisco JTAPI Extension Classes  A-23
Preface

This chapter introduces Cisco JTAPI implementation, describes the purpose of this document, and outlines the required software. The following topics are included:

- Introduction
- Purpose
- Audience
- New and Changed Information
- Organization
- Related Documentation
- Required Software
- Conventions
- Obtaining Documentation
- Obtaining Technical Assistance
Introduction

Java Telephony Application Programming Interface (JTAPI) is a portable, object-oriented API for computer telephony integrated call control. The package of JTAPI interfaces located in the java.telephony.* hierarchy, defines a programming model by which Java applications interact with telephony resources such as PBXs and telephones. The Cisco JTAPI implementation supports Java application access to Cisco Architecture for Voice, Video and Integrated Data (AVVID) communication systems according to the JTAPI v 1.2 specification. Furthermore, Cisco JTAPI exposes Cisco specific events and methods for certain telephony resources such as calls and connections.

Purpose

One of the primary goals of a standard Application Programming Interface (API) such as JTAPI is to provide an unchanging programming interface under which varied implementations may stand. Cisco's goal in implementing JTAPI for the Cisco CallManager platform is to conform as closely as possible to the JTAPI specification, while providing extensions that enhance JTAPI and expose the advanced features of Cisco CallManager to applications.

As new versions of Cisco CallManager and the Cisco JTAPI implementation are released, variances in the API should be very minor, and should tend in the direction of compliance. Cisco is committed to maintaining its API extensions with the same stability and reliability, though additional extensions may be provided as new Cisco CallManager features become available.

This document outlines some basic JTAPI concepts including transfer and conference extensions. It also describes the support of extensions to the Sun JTAPI v 1.2 specification.

Audience

This document is intended for telephony software developers who are developing Cisco IP Telephony applications that require JTAPI. This document assumes that the programmer is familiar with both the Java language and the Sun JTAPI v 1.2 specification.
New and Changed Information

This section describes any new features and or changes for Cisco JTAPI pertinent to the specified release of Cisco CallManager.

Cisco CallManager Release 3.3

The following lists the features or changes for Cisco JTAPI in Cisco CallManager release 3.3:

- **CallParkRequest, CallParkResponse, and Parked DN Monitoring**—defines new extensions to allow applications to park a call or unpark a call.
- **XSI Object Pass Through**—allows application to pass an XML object through a Cisco JTAPI or CTI interface to an IP phone.
- **Cisco JTAPI Installation Internationalization**—supports multiple languages for the Cisco JTAPI installation and the user preference user interface. Refer to the *Cisco CallManager 3.3 JTAPI Installation Guide* for more information.
- **Enable or Disable Ringer**—supports application control of ringer settings for each address on a device.
- **Clear Calls Interface**—provides a clearCallConnections interface that allows applications to remove phantom calls without removing the call observer.
- **Display Name Interface**—extends the CiscoCall interface to provide methods to get name displays of the calling party and the called party in a call. Applications can use getCurrentCallingPartyDisplayName() to get the display name of the calling party.
- **SetMessageWaiting Interface**—provides a method for applications the ability to set the message waiting lamp or indicator for an address.
- **Quite Clear**—provides QuiteClear at the other end when two parties are on a call and one address goes OutOfService because of a network outage, the Cisco CallManager goes down, application controlling CTIPort goes down, or CTIManager goes down.
- **GetCallInfo Interface**—provides applications the ability to query CallInfo on an address. A query returns the CiscoAddressCallInfo object, which contains information about the number of active or held calls, maximum number of active or held calls, and the Call object for current calls on the address. This interface also provides what calls are at a specific address at a specific time.
DeleteCall Interface—provides applications the ability to delete a call created using the createCall interface. This method accepts a call and throws an InvalidStateException if a provider is not in service or if the call is not in the IDLE state. DeleteCall moves the call to the INVALID state.

GetGCID—provides an interface on the CiscoCallID to get the nodeID and the GCID of the call; exposing the GCID information available in the internal call object.

GetCallID in RTP Events—provides an interface on RTP events to access any call information so that applications can link RTP events with the calls.

Cisco CallManager Release 3.2

The following lists the features or changes for Cisco JTAPI in Cisco CallManager release 3.2:

- Call Park—Cisco JTAPI supports user interactions with Call Park and reports appropriate events to the applications.
- Super Provider—supports static control of devices and the ability to query for devices.
- Separation of CTI client code—supports checking for provider properties in an object that implements the CtiClientProperties so application layers, other than Cisco JTAPI, can not utilize the functions in the client code to talk to the CTI Manager.
- Reconnect Logic—connects Cisco JTAPI applications to the secondary CTIManager after waiting for a random time so that all Cisco JTAPI applications do not connect to the secondary CTIManager at the same time.

Cisco CallManager Release 3.1

The following lists the features or changes for Cisco JTAPI in Cisco CallManager release 3.1:

- CTIManager component—allows a JTAPI application to control devices on another Cisco CallManager in a cluster. It supports multiple Cisco CallManagers and CTI managers during failover and recovery, and supports automatic device recovery during a failover.
• Directory Change Notification—allows asynchronous directory change notification.

• Transfer and Conference Enhancement—allows enhancements to Transferring & Conferencing.

• Call Forward Setting—allows Cisco JTAPI implementation for supporting Call Forwarding.

• CiscoJtapiExceptions—allows CiscoJtapiException handling modifications.

• Redirect—allows for a Cisco JTAPI Redirect request.

• Alarm Services—allows Cisco JTAPI support for Alarm Services.

• Application Control of JTAPI Parameters—allows control of the parameters within jtapi.ini.

• Dynamic Trace Enabling Using Jtprefs—allows dynamic enabling of traces from the Jtprefs application.

## Organization

The table below provides an outline of this document’s organization.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1, “Overview”</td>
<td>This chapter introduces the major concepts with which you need to be familiar before creating JTAPI applications for Cisco IP Telephony Solutions.</td>
</tr>
<tr>
<td>Chapter 2, “Cisco JTAPI Implementation”</td>
<td>This chapter describes the interfaces and classes available.</td>
</tr>
<tr>
<td>Chapter 3, “JTAPI Examples”</td>
<td>This chapter provides the source code for makecall, the Cisco JTAPI program used to test the JTAPI installation.</td>
</tr>
<tr>
<td>Appendix A, “JTAPI Classes and Interfaces”</td>
<td>This appendix contains a listing of all the classes and interfaces available in the Cisco JTAPI implementation for Cisco CallManager.</td>
</tr>
<tr>
<td>Appendix B, “CTI Error Codes”</td>
<td>This appendix lists and describes CTI error codes.</td>
</tr>
</tbody>
</table>
Related Documentation

The companion document Cisco JTAPI Developer Reference (JTAPI v 1.2 Specification) contains the Sun JTAPI specification for the supported interfaces and classes and their respective methods. The specification was downloaded from the Sun JTAPI web site and included in this document set as a convenience to Cisco JTAPI programmers. Note that the writing style in that book conforms to the Sun specification, it does not conform to the Cisco writing style.

To obtain the very latest version of the JTAPI specification files, go directly to the web site at:

- The Sun Microsystems Inc. JTAPI v 1.2 specification
  http://java.sun.com/products/jtapi/jtapi-1.2/packages.html

Required Software

The table below lists software requirements for the following applications: JTAPI applications, JTPREFS, and sample code.

<table>
<thead>
<tr>
<th>Application</th>
<th>Required Software</th>
<th>Examples</th>
</tr>
</thead>
</table>
| JTAPI applications | Any JDK 1.1 compliant java environment | • Microsoft Internet Explorer 4.01 or later  
|                  |                                    | • Sun JDK 1.1, 1.2, or 1.3        |
| JTPREFS         | Microsoft Internet Explorer 4.01 or later |                                 |
| Sample code     | Microsoft Internet Explorer 4.01 or later |                                 |
Conventions

This document uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong> font</td>
<td>Commands and keywords are in <strong>boldface</strong>.</td>
</tr>
<tr>
<td><em>italic</em> font</td>
<td>Arguments for which you supply values are in <em>italics</em>.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Elements in square brackets are optional.</td>
</tr>
<tr>
<td>{ x</td>
<td>y</td>
</tr>
<tr>
<td>[ x</td>
<td>y</td>
</tr>
<tr>
<td><em>screen</em> font</td>
<td>Terminal sessions and information the system displays are in <em>screen</em> font.</td>
</tr>
<tr>
<td><strong>boldface</strong> <em>screen</em> font</td>
<td>Information you must enter is in <strong>boldface</strong> <em>screen</em> font.</td>
</tr>
<tr>
<td><em>italic</em> <em>screen</em> font</td>
<td>Arguments for which you supply values are in <em>italic</em> <em>screen</em> font.</td>
</tr>
<tr>
<td>^</td>
<td>The symbol ^ represents the key labeled Control—for example, the key combination ^D in a screen display means hold down the Control key while you press the D key.</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Nonprinting characters, such as passwords are in angle brackets.</td>
</tr>
</tbody>
</table>

Notes use the following conventions:

*Note*  
Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.
Obtaining Documentation

These sections explain how to obtain documentation from Cisco Systems.

World Wide Web

You can access the most current Cisco documentation on the World Wide Web at this URL:

http://www.cisco.com

Translated documentation is available at this URL:


Documentation CD-ROM

Cisco documentation and additional literature are available in a Cisco Documentation CD-ROM package, which is shipped with your product. The Documentation CD-ROM is updated monthly and may be more current than printed documentation. The CD-ROM package is available as a single unit or through an annual subscription.

Ordering Documentation

You can order Cisco documentation in these ways:

- Registered Cisco.com users (Cisco direct customers) can order Cisco product documentation from the Networking Products MarketPlace:
  http://www.cisco.com/cgi-bin/order/order_root.pl
- Registered Cisco.com users can order the Documentation CD-ROM through the online Subscription Store:
  http://www.cisco.com/go/subscription
- Nonregistered Cisco.com users can order documentation through a local account representative by calling Cisco Systems Corporate Headquarters (California, U.S.A.) at 408 526-7208 or, elsewhere in North America, by calling 800 553-NETS (6387).
Documentation Feedback

You can submit comments electronically on Cisco.com. In the Cisco Documentation home page, click the Fax or Email option in the “Leave Feedback” section at the bottom of the page.

You can e-mail your comments to bug-doc@cisco.com.

You can submit your comments by mail by using the response card behind the front cover of your document or by writing to the following address:

Cisco Systems
Attn: Document Resource Connection
170 West Tasman Drive
San Jose, CA 95134-9883

We appreciate your comments.

Obtaining Technical Assistance

Cisco provides Cisco.com as a starting point for all technical assistance. Customers and partners can obtain online documentation, troubleshooting tips, and sample configurations from online tools by using the Cisco Technical Assistance Center (TAC) Web Site. Cisco.com registered users have complete access to the technical support resources on the Cisco TAC Web Site.

Cisco.com

Cisco.com is the foundation of a suite of interactive, networked services that provides immediate, open access to Cisco information, networking solutions, services, programs, and resources at any time, from anywhere in the world.

Cisco.com is a highly integrated Internet application and a powerful, easy-to-use tool that provides a broad range of features and services to help you with these tasks:

- Streamline business processes and improve productivity
- Resolve technical issues with online support
- Download and test software packages
Obtaining Technical Assistance

- Order Cisco learning materials and merchandise
- Register for online skill assessment, training, and certification programs

If you want to obtain customized information and service, you can self-register on Cisco.com. To access Cisco.com, go to this URL:

http://www.cisco.com

Technical Assistance Center

The Cisco Technical Assistance Center (TAC) is available to all customers who need technical assistance with a Cisco product, technology, or solution. Two levels of support are available: the Cisco TAC Web Site and the Cisco TAC Escalation Center.

Cisco TAC inquiries are categorized according to the urgency of the issue:

- Priority level 4 (P4)—You need information or assistance concerning Cisco product capabilities, product installation, or basic product configuration.
- Priority level 3 (P3)—Your network performance is degraded. Network functionality is noticeably impaired, but most business operations continue.
- Priority level 2 (P2)—Your production network is severely degraded, affecting significant aspects of business operations. No workaround is available.
- Priority level 1 (P1)—Your production network is down, and a critical impact to business operations will occur if service is not restored quickly. No workaround is available.

The Cisco TAC resource that you choose is based on the priority of the problem and the conditions of service contracts, when applicable.

Cisco TAC Web Site

You can use the Cisco TAC Web Site to resolve P3 and P4 issues yourself, saving both cost and time. The site provides around-the-clock access to online tools, knowledge bases, and software. To access the Cisco TAC Web Site, go to this URL:

http://www.cisco.com/tac
All customers, partners, and resellers who have a valid Cisco service contract have complete access to the technical support resources on the Cisco TAC Web Site. The Cisco TAC Web Site requires a Cisco.com login ID and password. If you have a valid service contract but do not have a login ID or password, go to this URL to register:

http://www.cisco.com/register/

If you are a Cisco.com registered user, and you cannot resolve your technical issues by using the Cisco TAC Web Site, you can open a case online by using the TAC Case Open tool at this URL:

http://www.cisco.com/tac/caseopen

If you have Internet access, we recommend that you open P3 and P4 cases through the Cisco TAC Web Site.

Cisco TAC Escalation Center

The Cisco TAC Escalation Center addresses priority level 1 or priority level 2 issues. These classifications are assigned when severe network degradation significantly impacts business operations. When you contact the TAC Escalation Center with a P1 or P2 problem, a Cisco TAC engineer automatically opens a case.

To obtain a directory of toll-free Cisco TAC telephone numbers for your country, go to this URL:


Before calling, please check with your network operations center to determine the level of Cisco support services to which your company is entitled: for example, SMARTnet, SMARTnet Onsite, or Network Supported Accounts (NSA). When you call the center, please have available your service agreement number and your product serial number.
Overview

This chapter introduces the major concepts with which you need to be familiar before creating JTAPI applications for Cisco IP Telephony Solutions. The following concepts are covered:

- Cisco JTAPI
- CiscoObjectContainer
- JtapiPeer and Provider
- Addresses and Terminals
- Threaded CallBacks
- Alarm Services
- Application Control of JTAPI Parameters
- Call Forward Setting
- Call Park
- CiscoJtapiExceptions
- Device Recovery
- Directory Change Notification
- Transfer and Conference Extensions
- Media Termination Extensions
- Redirect
- Routing
- Redundancy
Cisco JTAPI

An abstract telephony model, Java Telephony application Programming Interface (JTAPI) is capable of uniformly representing the characteristics of a wide variety of telecommunication systems. Because it is defined without direct reference to any particular telephony hardware or software, JTAPI is well suited to the task of controlling or observing nearly any telephone system. For instance, a computer program that makes telephone calls using an implementation of JTAPI for modems might work without modification using the Cisco JTAPI implementation.

As powerful as an abstraction may be in theory, in practice, programmers often need to know the details of how the JTAPI model represents the underlying components of a particular telephony system. A modem is a very simple telephony device with limited features. In contrast, the Cisco IP Telephony Solutions product family offers its users a comprehensive list of features and configuration capabilities. Programmers can best leverage the rich features of Cisco IP Telephony Solutions systems by understanding how it fits into the Cisco JTAPI model.

The following sections outline the deviations, if any, from the JTAPI v 1.2 specification or the specifics of the Cisco JTAPI implementation. Cisco JTAPI also provides extensions to the JTAPI v 1.2 specification. These extensions offer additional functionality to the Cisco implementation that is not defined in the specification. The classes and interfaces for the extensions are packaged in the com.cisco.jtapi.extensions package, and are explained in detail in Chapter 2, “Cisco JTAPI Extensions.”

CiscoObjectContainer

The CiscoObjectContainer interface allows applications to associate an application defined object to objects that implement this interface. In Cisco JTAPI, the following interfaces extend the CiscoObjectContainer interface:

- CiscoJTAPIPeer
- CiscoProvider
- CiscoCall
- CiscoAddress
- CiscoTerminal
JtapiPeer and Provider

The main point of contact between applications and JTAPI implementations is the Provider object, created through the implementation of the JtapiPeer object. The Provider object is a repository that contains the entire collection of call model objects, Addresses, Terminals, and Calls, controllable at any time by an application.

The JtapiPeer.getServer(), which returns server names, is administered by the JTPREFS applications.

The Provider entails two basic processes: initialization and shutdown.

The following information must be passed in the JtapiPeer.getProvider() method for applications to obtain a CiscoProvider:

- Hostname or IP address for the Cisco CallManager server
- Login of user administered in the directory.
- Password of user specified.
- (Optional) Application information—this parameter may be a string of any length.

Applications should include enough descriptive information so that if the appinfo were logged in an alarm, administrators would know which application caused the alarm. Applications should not include hostname or IP address where they reside, nor the time at which they were spawned. This information is either redundant with what JTAPI already logs or is potentially confusing when trying to track the progress of an application that is reconnecting on occasion. Also, no "=" or ";" characters are allowed in the appinfo string, as they are used to delimit the getProvider() string. When the appinfo is not specified, you can use a generic and quasi-unique name (JTAPI[XXXX]@hostname, where XXXX is a random four digit number) instead.
The parameters are passed in key value pairs concatenated in a string as follows:

```java
JtapiPeer.getProvider(
    "CTIManagerHostname;login=user;passwd=userpassword;appinfo=Cisco Softphone"
)
```

**Initialization**

The JtapiPeer.getProvider() method returns a Provider object as soon as the TCP link, the initial handshake with the Cisco CallManager, and device list enumeration are complete. The provider is now in the OUT_OF_SERVICE state. Cisco JTAPI applications must wait for the provider to go to the IN_SERVICE state before the controlled device list is valid. A ProvInServiceEv event is delivered to an object implementing the ProviderObserver interface. A upon an orderly Cisco CallManager shutdown.

**Note**

It is not sufficient to implement only the CiscoProviderObserver, the observer must also be added to the provider with provider.addObserver(...). Applications must wait for a notification that the Provider is in service.

**Shutdown**

When an application calls provider.shutdown(), JTAPI loses communications permanently with the Cisco CallManager, and a ProvShutdownEv event is delivered to the application. The application can assume that the Provider will not come up again and the application must handle a complete shutdown.

**Provider.getTerminals()**

This method returns an array of terminals created for the devices administered in the user control list in the directory. Refer to the Cisco CallManager Administration Guide to administer the user control list.
Provider.getAddresses()

This method returns an array of addresses created from the lines assigned to the devices administered in the user control list in the directory.

Changes to the User Control List in the Directory

If a device is added to the user control list after the JTAPI application starts, a CiscoTermCreatedEv, and the respective CiscoAddrCreatedEv, are generated and sent to observers implementing the CiscoProviderObserver. In addition, applications can monitor the current registration state of the controlled devices, and dynamically track the availability of these devices. The events for an in-service Address or Terminal are delivered to observers implementing the CiscoAddressObserver and the CiscoTerminalObserver.

Note

It is not sufficient to implement only the observers, the observers must also be added by address.addObserver(..) and, similarly, for the terminal by the terminal.addObserver(..) method.

Note

Before invoking the call.connect() method, add a CallObserver to the address or terminal originating the call. Otherwise, the method returns an exception.

Addresses and Terminals

The Cisco IP Telephony Solutions architecture has three fundamental types of endpoints: telephone sets, virtual devices (media termination points and route points), and gateways. Of these endpoints, only telephones and media termination points are exposed through the Cisco JTAPI implementation.

Cisco CallManager allows users to configure telephones to have one or more lines, dialable numbers, which may be shared among multiple telephones simultaneously or configured for exclusive use by only one telephone at a time. Each line on a telephone is capable of terminating two calls simultaneously, one of which must be on hold. This is similar to the operation of the “call waiting”
feature on home telephones. Figure 1-1 “Telephone Diagram” shows two configurations, Peter and Mary are sharing one telephone line, 5001, while Paul is configured with his own telephone line, 5002.

Figure 1-1 Telephone Diagram

"Peter" 5000

5001

"Mary" 5001

"Paul" 5002

Address and Terminal Relationship

All types of Cisco CallManager endpoints are identified by a unique name. A telephone terminal is identified by its hardware Media Access Control (MAC) address (e.g. “SEP0010EB1014”), whereas the system administrator may assign any name to a media termination point, so long as its name is distinct. For each endpoint controlled by a Provider, the Cisco JTAPI implementation uses its administered name to construct a corresponding Terminal object. Terminal objects in turn have one or more Address objects, each of which corresponds to a line on the endpoint. Figure 1-2 “Address and Terminal Relationship” shows a graphical representation of the relationship between Addresses and Terminals.
Figure 1-2  Address and Terminal Relationship

If two or more endpoints share a line (directory number), then the corresponding Address object will be related to more than one Terminal object.

Note  Cisco JTAPI does not support shared lines.

Unobserved Addresses and Terminals

Cisco JTAPI learns about calls only when there is a CallObserver attached to the terminals/addresses of the Provider. This means that methods such as Provider.getCalls() or Address.getConnections() will return null, even when there are calls at the address, unless there is a CallObserver attached to the address. It is also required to add a CallObserver to the Address or Terminal originating a call via the Call.connect(..) method.

Connection

Connections retain their references to calls and addresses forever. So, a connection reference obtained from a call event can always be used to obtain the connection call (getCall()) and address (getConnection()).
TerminalConnection

TerminalConnections always retain their references to terminals and connections. So, a terminal connection reference obtained from a call event can always be used to obtain the terminal connection terminal (getTerminal()) and connection (getConnection()).

CiscoConnectionID

The CiscoConnectionID object represents a unique object associated with each connection in Cisco JTAPI. Applications may use the object itself or the integer representation of the object.

CiscoCallID

The CiscoCallID object represents a unique object associated with each connection in Cisco JTAPI. Applications may use the object itself or the integer representation of the object.

Threaded CallBacks

The Cisco JTAPI implementation is designed to allow applications to invoke blocking JTAPI methods such as Call.connect() and TerminalConnection.answer() from within their observer callbacks. This means that applications are not subject to the restrictions imposed by the JTAPI 1.2 specification, which cautions applications against using JTAPI methods from within observer callbacks. Applications, however, can selectively disable the queuing logic of the Cisco JTAPI implementation by implementing the CiscoSynchronousObserver interface on their observer objects.

Many applications are not adversely affected by this asynchronous behavior. Applications that would benefit from a coherent call model during observer callbacks, however, can selectively disable the queuing logic of the Cisco JTAPI.
implementation. By implementing the CiscoSynchronousObserver interface on its observer objects, an application declares deliver synchronous events to its observers. Events delivered to synchronous observers will match the states of the call model objects queried from within the observer callback.

**Note**

Objects that implement the CiscoSynchronousObserver interface are strictly forbidden from invoking blocking JTAPI methods from within their event callbacks. The consequences of doing so are unpredictable, and may include deadlocking the JTAPI implementation. However, they may safely use the accessor methods of any JTAPI object, for instance Call.getConnections() or Connection.getState().

### Querying Dynamic Objects

Beware of querying dynamic objects such as call objects. By the time you get an event, the object (such as, call) may be in a different state than the state indicated. For example, by the time you get a CiscoTransferStartEV, the transferred call, may have removed all of its internal connections.

**callChangeEvent()**

When the callChangedEvent() method is called, the validity is guaranteed for any references contained in the event. For example, if the event contains a getConnection() method, then the application can call this method and get a valid connection reference. Likewise, a getCallingAddress() method is guaranteed to return a valid Address object.

**CiscoConsultCall**

For the CiscoConsultCall interface, a reference to a consulting terminal connection is retained forever. For example, when processing a CiscoConsultCallActive event, getConsultingTerminalConnection() is guaranteed to return a valid terminal connection reference. Further, the terminal connection is guaranteed to provide access to the consulting connection and thus the consulting call.
CiscoTransferStartEv

For the CiscoTransferStartEv, the references to the transferred call, transfer controller and final call in the event will be valid when callChangedEvent() is called. However, the connections on these calls may or may not still be returned by getConnections().

Alarm Services

Support for sending alarms to a service is included as part of the general serviceability framework for Cisco IP telephony applications. The alarm components are defined in the com.cisco.services.alarm package.

An alarm interface and framework supports the sending of alarm notifications in XML over TCP to an Alarm Service available on the network in a Cisco JTAPI applications. The alarm package features include:

- XML definition of alarms, resolved by a catalog in the alarm service.
- A bounded rollover queue to buffer alarms at the sender.
- Alarm sending on a separate thread to avoid blocking at the sending application.
- A TCP-based reconnection scheme to the alarm service.

The overall framework of the Cisco JTAPI alarm system is similar to the existing JTAPI tracing package. Applications must instantiate an AlarmManager for a particular facility code from which alarm objects can be created. DefaultAlarm and DefaultAlarmWriter implementation classes are included as part of the implementation.

Application Control of JTAPI Parameters

The various parameters required for configuring Cisco JTAPI are located in the jtapi.ini file. Cisco JTAPI looks for the presence of this file in the current Java classpath.
Cisco JTAPI also supports application setting of all the parameters required by Cisco JTAPI. This allows a single point of application administration, independent of jtapi.ini. The jtapi.ini file is a starting point for default values but client applications can modify different values without having to specifically modify the jtapi.ini file.

Applications obtain a CiscoJtapiProperties object from the CiscoJtapiPeer and make changes to the parameters using the accessor and mutator methods. These properties apply peer-wide to all the providers derived from a CiscoJtapiPeer and must be set prior to the first getProvider() call on that peer.

Different instances of client applications, however, can impose different settings for these parameters. The CiscoJtapiProperties interface is defined in the com.cisco.jtapi.extensions package.

**Jtapi.ini Parameters**

Cisco JTAPI is configured using parameters in the jtapi.ini. These parameters are modified using the Jtprefs application, which is installed by the Cisco JTAPI installer.

The following parameters are pertinent to Cisco JTAPI:

```
#Cisco Jtapi version 1.4(1.7) Release .ini parameters
#Wed Oct 30 10:53:04 PST 2002
PROTOCOL_DEBUGGING=1
UseSameDirectory=1
JTAPIIMPL_DEBUGGING=1
UseSystemDotOut=0
QueueStatsEnabled=0
PeriodicWakeupInterval=50
RouteSelectTimeout=5000
UseTraceFile=1
ProviderOpenRequestTimeout=200
CtiManagers=cm-server1;cm-server2
Directory=SEAVIEWTraces
```
DEBUG=1
DesiredServerHeartbeatInterval=30
AlarmServicePort=1444
CTI_DEBUGGING=1
SyslogCollector=
JTAPI_DEBUGGING=1
PeriodicWakeupEnabled=0
NumTraceFiles=10
AlarmServiceHostname=
MISC_DEBUGGING=1
TracePath=. 
UseAlarmService=0
CTIIMPL_DEBUGGING=1
WARNING=1
Traces=WARNING;INFORMATIONAL;DEBUG
INFORMATIONAL=1
UseSyslog=0
JtapiPostConditionTimeout=15
JTAPINotificationPort=789
FileNameBase=seaview
CtiRequestTimeout=30
TraceFileSize=1048576
Debugging=JTAPI_DEBUGGING;JTAPIIMPL_DEBUGGING;CTI_DEBUGGING;CTIIMPL_DEBUGGING;PROTOCOL_DEBUGGING;MISC_DEBUGGING
FileNameExtension=log
QueueSizeThreshold=25
ProviderRetryInterval=30
SyslogCollectorUDPPort=514
Dynamic Trace Enabling Using Jtprefs

You will generally want to turn off high level debugging traces. Typically, tracing at full debug levels imposes a load on the system and is not desirable during normal system operation. For troubleshooting purposes, an administrator can turn tracing on or off dynamically.

Cisco JTAPI supports the dynamic enabling of traces from the Jtprefs administrative application. The application communicates with JTAPI using a TCP socket and sends a signal to enable or disable the traces. For more details on dynamic tracing, refer to the Cisco CallManager Administration Guide. It is necessary for the trace file generation to be turned on prior to using dynamic tracing. If no trace files are being generated, dynamic tracing does not enable file generation. Dynamic tracing only enables or disables traces on an existing log file stream. By default, JTAPI enables file writing with all tracing levels off.

Call Forward Setting

Cisco JTAPI supports setting the Call Forward feature according to the JTAPI Specification. Cisco's JTAPI implementation does not support all the forwarding characteristics but supports only the FORWARD_ALL attribute for the Address. Applications can invoke setForwarding, getForwarding and cancelForwarding methods on a CallControlAddress object, but the CallControlForwarding instruction can only be of type FORWARD_ALL.

Call Park

Cisco JTAPI supports user interactions with Call Park and reports the appropriate events to the applications. When a call is parked from an IP phone, the connection that belongs to the parking address moves into Disconnected state the associated TerminalConnection moves into Dropped state. A new connection in queued state for the park number is created.

If an application is monitoring only the address that parked the call, then all existing connections are Disconnected, TerminalConnections are Dropped, and the call moves to Invalid state.
Park Retrieval

When a call is parked from an IP phone, the park number is displayed on the phone. Any terminal can unpark the call by dialing the park number. When a call is unparked, a new call is created with connections to unparked address. The CallControlConnection for the park number in the original call, which is in the Queued state, moves to the Disconnected state.

Park Reminder

When a parked call is not retrieved for a specified time, a reminder call returns to the address that parked the call and Park Number connection moves to the Disconnected state. The call reconnects and moves to the Established state. A terminal connection in Talking state is created for the address that parked the call.

Park DN Monitor

CTI applications using Cisco JTAPI client can start monitoring the call park DNs using the EnumerateParkDn interface. This returns all the park DN lines in the system. Open all the lines to receive parking activity. Applications receive Existing Call, New Call, and CallStateChanged events on the lines.

To stop monitoring the park DNs, applications can close all the opened park DN lines.

Public interface ICCNProvider

```java
{
    public boolean canMonitorParkDNs();
    public java.util.Enumeration EnumerateParkDns () throws CCNException;
    public OpenParkDns( ICCNLineEvents) throws CCNException
    public CloseParkDns() throws CCNException
}
```
CiscoJ tapiExceptions

Cisco JTAPI notifies application of CTI-generated error codes. These codes are returned when there is an exception or error in the CTIManager. The CTI returned error code is propagated to the application separately. The application can extract the error code by invoking getErrorCode() method on the exception object, can get CTI error code name by invoking getErrorName() method and can get error description by invoking method getErrorDescription(). The methods getErrorName(int errorCode) and getErrorDescription (int errorCode) are deprecated and will be removed in future releases. Application users are discouraged to use these.

Cisco J TAPI Install Internationalization

Cisco JTAPI supports multiple languages for the JTAPI installation and user preference UI. When JTAPI launches, you are given options for selecting languages for the installation. After selecting a language, further installation instructions are in the selected language. The first option is always English. If certain phrases are missing in the locale language, the instructions default to English. Refer to the Cisco JTAPI Installation Guide for more installation information.

Clear Calls Interface

Cisco JTAPI applications can clear phantom calls without dropping active calls. The CiscoAddress provides a clearCallConnections message to allow applications to clear the calls when there are no active calls on the Cisco CallManager.

Device Recovery

Automatic device recovery is supported as part of Cisco JTAPI.
Device Recovery for phones

For devices such as the Cisco IP Phone 7960, the re-homing feature is part of the device firmware. On a primary Cisco CallManager failure, the phone attempts to connect to the backup Cisco CallManager when it is no longer on a call. This transition is communicated to applications in the form of out-of-service and in-service events described in the CTIManager Failure section.

For virtual devices with no firmware such as CTI Ports and CTI RoutePoints, the failover is performed by the CTIManager or by Cisco JTAPI.

CTI RoutePoints

On a Cisco CallManager server failure, the CTIManager recovers the device from the Cisco CallManager server group as defined in the device pool administration for the CTI RoutePoint. When the primary Cisco CallManager server recovers, the CTIManager attempts to recover the device on its primary Cisco CallManager. This re-homing happens when there are no more calls on the device, (similar to physical devices). On a CTIManager failure, Cisco JTAPI recovers the device on the backup CTIManager. The application is informed of the availability of a device with the CiscoAddrOutOfServiceEv and CiscoAddrInServiceEv events.

CTI Ports

CTI Ports that are registered by an application have a mechanism similar to phone devices. When the Cisco CallManager that is handling signaling for a CTI Port fails, the CTIManager recovers its services according to the device pool administration for this device. On a CTIManager failure, Cisco JTAPI re-registers the CTI Port after it has connected to the backup CTIManager. The CiscoAddrOutOfServiceEv and CiscoTermOutOfServiceEv events and the corresponding in service events are sent after recovery of the CTI Port.

Media streaming for these devices is controlled by the application and continues even when the port is out of service. It is the application's responsibility to ensure that new calls do not get presented to the device until it is ready to accept them.
Directory Change Notification

Applications are notified asynchronously of device additions or deletions from the user control list and device deletions from the Cisco CallManager database. Applications also receive notification about line changes to a device. This notification is sent to Cisco JTAPI and is propagated to applications with CiscoAddrCreatedEv, CiscoAddrRemovedEv, CiscoTermCreatedEv, and CiscoTermRemovedEv on the AddressObserver and TerminalObservers respectively.

Note

The device needs to be registered for CTIPorts and CTIRoutePoints to receive the line change notification.

Enable or Disable Ringer

The CiscoAddress extension allows applications to set the status of the ringer for all lines on a device. No events are generated when the ringer setting changes from the administration pages or anywhere else.

Transfer and Conference Extensions

Transfer and Conference events are difficult to understand in JTAPI. The reason is that when the participants are moved from one call to the other, JTAPI represents this by deleting the parties from one call and adding them to the other call. It may be confusing for an application to receive an indication that a party has dropped from the call when in reality it is in the process of being moved. The Cisco JTAPI implementation defines some extra events that make it easier for applications to deal with these functions.

Transfer

The transfer feature moves the participants of one call, the transferred call, to another call, the final call. Moving participants in a call results in their associated Connections transitioning to the DISCONNECTED state in the transferred call.
and new Connections for these participants being created in the final call. Similarly, any associated TerminalConnections transition into the DROPPED state in the transferred call and are created in the final call. Cisco extensions have been defined to mark the start and the end of the events that are related to transfer.

One way to correlate the newly created connection objects with the old connection objects is to use the CiscoConnection.getConnectionID() method to obtain the CiscoConnectionID for the old and new connections. Matching connections will have identical CiscoConnectionID objects when compared using the CiscoConnectionID.equals() method.

**CiscoTransferStartEv**

This event indicates that the transfer operation has started, and the events that follow are related to this operation. Specifically, Connections and TerminalConnections are both removed and added as a result of the transfer. Applications may obtain the two calls involved in transfer—transferred call and final call, and the transfer controller information from this event. If the transfer controller is not being observed by the JTAPI application, the transfer controller information is not available in this event.

**CiscoTransferEndEv**

This event indicates that the transfer operation has ended. After this event is received, the application can assume that all involved parties have been transferred, and all Connections and TerminalConnections have been moved to the final call.

**Transfer Scenarios**

In the following scenarios, A, B, and C are three parties involved in the transfer.

**Blind Transfer, Where B is the Transfer Controller**

In blind transfer, a call that has been answered already is redirected without issuing separate consult and transfer commands.

- A calls B on call Call1
- B answers and blind transfers the call to C
Chapter 1      Overview

Transfer and Conference Extensions

In this scenario, a consult call (Call2) is placed internally between B and C. To do this type of transfer the following JTAPI method is necessary:

- Call1.transfer (C)

Table 1-1 lists the core events that observers for A and B receive between the CiscoTransferStartEv and the CiscoTransferEndEv:

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_UNKnown</td>
<td>Call1</td>
<td>CiscoTransferStartEv</td>
<td>transferredCall=Call2</td>
</tr>
<tr>
<td>META_UNKnown</td>
<td></td>
<td>CallObservationEndedEv</td>
<td>finalCall=Call1</td>
</tr>
<tr>
<td>META_UNKnown</td>
<td></td>
<td>CiscoTransferEndEv</td>
<td>transferController=TermConnB</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call1</td>
<td>ConnCreatedEv</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>ConnAlertingEv</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>ConnDisconnectedEv</td>
<td>FinalCall=Call1</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>CallCtlConnDisconnectedEv</td>
<td>CallCtlTermConnDroppedEv B</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td></td>
<td>ConnDisconnectedEv B</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv B</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call2</td>
<td>ConnCreatedEv</td>
<td>transferredCall=Call2</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>ConnAlertingEv</td>
<td>finalCall=Call1</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>ConnDisconnectedEv</td>
<td>transferController=TermConnB</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td>CallCtlConnDisconnectedEv</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1-1 Core Events for Observers for A and B
Consult Transfer, Where B is the Transfer Controller

In a consult transfer, applications can redirect calls to a different address and the transferrer can “consult” with the transfer destination before redirecting.

- A calls B on call Call1
- B answers and consults to C on call Call2
- B transfers call Call2 to call Call1

To do this type of transfer the following JTAPI methods are necessary:

- Call2.setTransferEnable(true) (This is optional because transfer is enabled in the call object by default.)
- Call2.consult(TermConnB, C)
- Call1.transfer(Call2)

Table 1-2 lists the core events that observers of A and B receive between the CiscoTransferStartEv and the CiscoTransferEndEv:

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_UNKNOWN</td>
<td>Call1</td>
<td>CiscoTransferStartEv</td>
<td>transferredCall=Call2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>finalCall=Call1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>transferController=TermConnB</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call1</td>
<td>TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B</td>
<td>transferredCall=Call2 finalCall=Call1 transferController=TermConnB</td>
</tr>
</tbody>
</table>

Note: During consult transfer, Call1.transfer(Call2) will transfer the call, but not Call2.transfer(Call1).
Transfer and Conference Extensions

Chapter 1      Overview

Arbitrary Transfer, where A is the Transfer Controller

In an arbitrary transfer, one call can be transferred to another call, irrespective of how either call was created. Unlike consult transfer, there is no need to first create one of the calls using the consult method.

- A calls B on call Call1
- A puts Call1 on hold
- A calls C on call Call2
- A transfers Call1 to Call2

To do this type of transfer the following JTAPI methods are necessary:

- Call2.transfer(Call1) to transfer call Call1 to final call Call2, or

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_CALL_TRANSFER RING</td>
<td>Call1</td>
<td>ConnCreatedEv C ConnConnectedEv C CallCtlConnEstablishedEv C TermConnCreatedEv C TermConnActiveEv C CallCtlTermConnTalkingEv C</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER RING</td>
<td>Call2</td>
<td>TermConnDroppedEv B CallCtlTermConnDroppedEv B ConnDisconnectedEv B CallCtlConnDisconnectedEv B</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER RING</td>
<td>Call2</td>
<td>TermConnDroppedEv C CallCtlTermConnDroppedEv C ConnDisconnectedEv C CallCtlConnDisconnectedEv C CallInvalidEv C</td>
<td></td>
</tr>
<tr>
<td>META_UNKNOWN</td>
<td>Call2</td>
<td>CallObservationEndedEv</td>
<td></td>
</tr>
<tr>
<td>META_UNKNOWN</td>
<td>Call1</td>
<td>CiscoTransferEndEv</td>
<td>transferredCall=Call2 FinalCall=Call1 TransferController=TermConnB</td>
</tr>
</tbody>
</table>

Table 1-2   Core Events for Observers of A and B (continued)
· Call1.transfer(Call2) to transfer call Call2 to final call Call1

Assuming Call1.transfer(Call2) was called, Table 1-3 lists the core events observers on A and C receive between CiscoTransferStartEv and CiscoTransferEndEv:

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_UNKNOWN</td>
<td>Call1</td>
<td>CiscoTransferStartEv</td>
<td>transferredCall=Call2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>finalCall=Call1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>transferController=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TermConnB</td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call1</td>
<td>TermConnDroppedEv B</td>
<td></td>
</tr>
<tr>
<td>RING</td>
<td></td>
<td>CallCtlTermConnDroppedEv B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnDisconnectedEv B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv B</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call1</td>
<td>ConnCreatedEv C</td>
<td></td>
</tr>
<tr>
<td>RING</td>
<td></td>
<td>ConnConnectedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnEstablishedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TermConnCreatedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TermConnActiveEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlTermConnTalkingEv C</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call2</td>
<td>TermConnDroppedEv B</td>
<td></td>
</tr>
<tr>
<td>RING</td>
<td></td>
<td>CallCtlTermConnDroppedEv B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnDisconnectedEv B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv B</td>
<td></td>
</tr>
<tr>
<td>META_CALL_TRANSFER</td>
<td>Call2</td>
<td>TermConnDroppedEv C</td>
<td></td>
</tr>
<tr>
<td>RING</td>
<td></td>
<td>CallCtlTermConnDroppedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnDisconnectedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallInvalidEv C</td>
<td></td>
</tr>
</tbody>
</table>

**Conference**

When conferencing two calls together, JTAPI specifies that all the parties from one call be moved to the other call. The call whose parties are moved away and which becomes invalid subsequently is called the “merged” or “consult” call
hereafter. The call to which the merged parties move is called the “final” call hereafter. When parties move from the merged call to the final call, the application receives events indicating that all parties have dropped from the merged call, and events indicating that the parties have reappeared on the final call.

One way to correlate the newly created connection objects with the old connection objects is to use the CiscoConnection.getConnectionID() method to obtain CiscoConnectionID objects for all old connections and all new connections. Matching connections will have identical CiscoConnectionID objects when compared using the CiscoConnectionID.equals() method.

Conference support exists for:

- `javax.telephony.callcontrol.CallControlCall.conference(Call)`
- `javax.telephony.callcontrol.CallControlCall.getConferenceController()`
- `javax.telephony.callcontrol.CallControlCall.getConferenceEnable()`
- `javax.telephony.callcontrol.CallControlCall.setConferenceController(TerminalConnection)`
- `javax.telephony.callcontrol.CallControlCall.setConferenceEnable(boolean)`

Cisco Extensions

Cisco JTAPI implementation provides two extra events that signal the Start and End of Conference: CiscoConferenceStartEv and CiscoConferenceEndEv. These events are sent when Conference is initiated and when it is completed. They give handles to the final call, the merged conference (consult) call and the two controlling TerminalConnections (in HELD and TALKING state).

CiscoConferenceStartEv

This event is sent when call1.conference(call2) is invoked or if the Conference button is pressed for the second time on an IP Phone. The ConferenceStartEv signifies the start of the merging process. This event is followed by a sequence of merging events reflected by the Conference process in Cisco CallManager described in the previous section.
CiscoConferenceEndEv

This event is sent at the end of the merge process after a ConferenceStartEv is sent. It signifies the completion of the merge of the Consult (or Merged) call into the Final Conference Call. The Merged call is in INVALID state and an ObservationEndedEv is sent for the call observer.

CiscoCall.setConferenceEnable()

The Cisco JTAPI implementation uses the CiscoCall.setConferenceEnable() and the CiscoCall.setTransferEnable() methods to control whether the consult call will be initiated via the conference or the transfer feature. If none of the features are enabled explicitly, transfer will be used by default.

Conference Scenarios

The following scenarios describe the two typical types of Conference that can be invoked.

Consult Conference with B as the Conference Controller

The following sequence of steps typically describe this scenario:

- A calls B (Call 1)
- B answers
- B Consults C (Call 2)
  
  ```
  setConferenceEnable()
  call2.consult(tc, C)
  ```
- C answers
- B Completes Conference
  
  ```
  Call1.conference(Call2)
  ```

Note: You must invoke the conference() method on the original call to complete a conference after a consultation. Invoking conference in the consult call object throws an exception.
Arbitrary Conference with B as the Conference Controller

The following sequence of steps typically describe this scenario:

• A calls B (Call 1)
• B answers
• B places the call on hold
• B calls C (Call 2)
• C answers
• B Completes Conference

Call1.conference(Call2) or
Call2.conference(Call1)

Conference Events

Table 1-4 provides the sequence of Core, Call control and Cisco Extension events when Call1.Conference(Call2) is called:

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_UNKNOWN</td>
<td>Call1</td>
<td>CiscoConferenceStartEv</td>
<td>consultCall = Call2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>finalCall = Call1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>conferenceController=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TermConnB</td>
</tr>
<tr>
<td>META_CALL_MERGING</td>
<td>Call1</td>
<td>CallCtlTermConnTalkingEv</td>
<td>B</td>
</tr>
<tr>
<td>META_CALL_MERGING</td>
<td>Call1</td>
<td>ConnCreatedEv</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnConnectedEv</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnEstablishedEv</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TermConnCreatedEv</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TermConnActiveEv</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlTermConnTalkingEv</td>
<td>C</td>
</tr>
<tr>
<td>META_CALL_MERGING</td>
<td>Call2</td>
<td>TermConnDroppedEv</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlTermConnDroppedEv</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnDisconnectedEv</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv</td>
<td>B</td>
</tr>
</tbody>
</table>
Transfer and Conference Extensions

Table 1-4  Sequence of Events (continued)

<table>
<thead>
<tr>
<th>Meta Event Cause</th>
<th>Call</th>
<th>Event</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>META_CALL_MERGING</td>
<td>Call2</td>
<td>TermConnDroppedEv C</td>
<td>consultCall=Call2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlTermConnDroppedEv C</td>
<td>finalCall=Call1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ConnDisconnectedEv C</td>
<td>conferenceController=TermConnB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallCtlConnDisconnectedEv C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CallInvalidEv</td>
<td></td>
</tr>
<tr>
<td>METAUNKNOWN</td>
<td>Call2</td>
<td>CallObservationEndedEv</td>
<td></td>
</tr>
<tr>
<td>METAUNKNOWN</td>
<td>Call1</td>
<td>CiscoConferenceEndEv</td>
<td></td>
</tr>
</tbody>
</table>

Transfer and Conference Enhancement

All parties involved in the call transfer in are sent CiscoTransferStartEv and CiscoTransferEndEv. All parties involved in the call conference are sent CiscoConferenceStartEv and CiscoConferenceEndEv. A call transfer still generates two events—the dropping of a connection to the first call and the creation of a connection to the second call. This order has been changed in Cisco CallManager Release 3.1. Connections are first created in the final call and then dropped in the consult call.

Note

Not All parties involved in the transfer of calls were delivered these events in Cisco CallManager Release 3.0

Note

Applications should not rely on the order of events between CiscoTransferStartEv and CiscoTransferEndEv or between CiscoConferenceStartEv and CiscoConferenceEndEv for transferring and conferencing when porting applications from Cisco CallManager Release 3.0 to 3.1.

Consult Without Media

Applications can inform Cisco CallManager that a consultation call for a transfer is being placed without establishing the media path. It is not necessary to establish the media path for the intermediate call, if the consultation call is being placed to
determine whether an agent is available before the actual transfer. The consultWithoutMedia method as defined in the CiscoConsultCall interface creates a consultation call without establishing the media path.

Note

The consultation call may only be transferred. It can not be conferenced.

Media Termination Extensions

Media termination is a feature that allows applications to transmit and capture the bearer of a call, for example, audio or video. This is sometimes referred to as “rendering and recording” or “sourcing and sinking” media. It is distinct from call control because media termination concerns the data that flows between endpoints in a call, not the details of setting up or tearing down calls. For example, an automatic call distributor (ACD) uses call control to route calls among available agents, but does not terminate media. An interactive voice response (IVR) application, on the other hand, uses call control to answer and disconnect calls and uses media termination to play sound files to callers.

Although there are no telephony applications that are solely interested in media termination, this feature is always used in combination with call control. JTAPI 1.2 is primarily a call control specification, and offers very limited support for applications that require media termination. Since the Cisco IP Telephony Solutions platform supports media termination to a much greater degree than JTAPI standard, the Cisco JTAPI implementation extends JTAPI to add full support for this feature.

In Cisco JTAPI, software-based media termination is accomplished using Computer Telephony Integration (CTI) Ports. They have one or more lines (dialable numbers) that can be used to originate or receive calls. They however need a controlling application to provide the source and sink of the media. An application registers its interest in the media termination port with the Cisco CallManager. The Cisco CallManager then delivers all the events related to this virtual device to the application. In Cisco JTAPI, CTI Ports are referred to as CiscoMediaTerminals. Figure 1-3 shows the CTI Port configuration. For details about administering and configuring a CTI Port, refer to the Cisco CallManager Administration web pages.
To implement a softphone application (where the PC acts as the telephone set, for example), the Cisco JTAPI application would manage a CTI Port.

Cisco CallManager Media Endpoint Model

Endpoints are the entities within the Cisco IP Telephony Solutions platform that terminate media, such as IP telephones and gateways. A call from one endpoint to another results in media flowing between the two endpoints. All endpoints in the Cisco IP Telephony Solutions platform transmit voice data using real-time protocol (RTP). The Cisco IP Telephony Solutions telephones and gateways, for example, have built-in RTP stacks. Applications may also act as endpoints in an Cisco IP Telephony Solutions system, that is, they may terminate media. Since all Cisco IP Telephony Solutions endpoints use RTP, applications must also be able to transmit and receive RTP packets.

Payload and Parameter Negotiation

In addition to bearer data, payload, each RTP packet contains a header that helps endpoints to determine how to reassemble and decode a sequence of such packets into a media stream. What RTP does not provide, however, is a means for endpoints to negotiate which payload type to use for a particular stream, for example, audio data encoded using the G.711 standard. Furthermore, RTP does not offer a means of negotiating unique payload type parameters such as the sampling rate of the encoded data or the number of samples that are to be transferred in each RTP packet. Instead, RTP is usually used in conjunction with another protocol such as H.323, which specifies its own method for endpoints to negotiate these parameters. All such negotiation is performed prior to transmitting RTP packets between endpoints.
Cisco CallManager, not the endpoints, is responsible for selecting the payload and encoding parameters for RTP streams. The five steps involved in a typical bidirectional audio telephone call are as follows:

- Initialization
- Payload Selection
- Receive Channel Allocation
- Starting Transmission and Reception
- Stopping Transmission and Reception

**Initialization**

Upon startup, each endpoint informs Cisco CallManager of its media capabilities, i.e., G.711, G.723, G.729a, etc. Startup for an IP telephone, for example, occurs when the telephone is first turned on, or after it recontacts Cisco CallManager after losing its former connection. The endpoint cannot express a preference for one payload type versus another, but it can specify certain parameters for each payload type, such as, packet size.

The capability list registered by the endpoint is exclusive and immutable. If the endpoint specifies that it can support both G.711 and G.723, it is implicitly declaring that it cannot support G.729a. Moreover, the endpoint must disconnect from Cisco CallManager and reinitialize in order to change the list of capabilities that it supports.

JTAPI applications perform this step by registering a CiscoMediaTerminal with Cisco CallManager. The CiscoMediaTerminal.register() method allows applications to supply an array of media capability objects for registration with Cisco CallManager. This step informs Cisco CallManager that the application will act as the endpoint for all calls to or from a particular directory number, as determined by the device configuration in the Cisco CallManager configuration.

**Payload Selection**

When a bidirectional media stream is about to be created between two endpoints, for instance, when a call is answered at an endpoint, Cisco CallManager selects an appropriate payload type (codec) for the media stream. Cisco CallManager compares the media capabilities of both endpoints involved in the call, and selects the appropriate common payload type and payload parameters to use. Payload
selection is based on endpoint capabilities and location, although there may be other criteria added to this selection logic in the future. The important thing to understand is that endpoints are not dynamically involved in selecting payload types on a call-by-call basis.

Receive Channel Allocation

If Cisco CallManager is able to find a common payload type for the RTP stream between the two endpoints, it requests that each endpoint create a logical “receive channel,” that is, a unique IP address and port at which the endpoint will receive RTP data for the call. Each endpoint returns an IP address and port to Cisco CallManager in response to this request.

Currently, only IP telephones and gateways perform this step. Cisco CallManager requires JTAPI applications to specify a fixed IP address and port during initialization. Therefore, JTAPI applications cannot terminate more than one media stream simultaneously for the same endpoint. Applications that wish to terminate multiple media streams must register multiple endpoints simultaneously.

If the endpoint does not respond to the open receive channel request quickly enough, Cisco CallManager disconnects the call. Because JTAPI applications always supply an IP address when registering CiscoMediaTerminals, calls to application-controlled endpoints will not be disconnected for this reason. However, if Cisco CallManager cannot find a common payload type between the two endpoints involved in the call, Cisco CallManager disconnects the call.

Starting Transmission and Reception

After Cisco CallManager receives channel information for both parties, it informs each endpoint of the codec parameters that it selected for the RTP stream and the destination address for the other endpoint. This information is conveyed in two messages to each endpoint: a start transmission message and a start reception message.

JTAPI applications receive the CiscoRTPOutputStartedEv and CiscoRTPInputStartedEv events that contain all of the codec parameters necessary for sending and receiving RTP data.
Stopping Transmission and Reception

When the RTP stream must be interrupted because of a feature such as hold or disconnect, Cisco CallManager requests that each endpoint stop its transmission and reception of RTP data. Just as when the media flow is started, the stop transmission and stop reception messages are sent separately.

JTAPI applications receive the CiscoRTPOutputStoppedEv and CiscoRTPInputStoppedEv.

CiscoMediaTerminal

In JTAPI, the terminal object represents the logical endpoint for a call and is presumed to be able to receive and transmit data (digitally-encoded voice samples, for example). Thus, Cisco IP Phones are represented by terminals in JTAPI. Even though gateways terminate media, they are not represented by terminals, however. The CiscoMediaTerminals in particular represent a special kind of endpoint for which applications take responsibility for media termination.

There are four steps associated with using CiscoMediaTerminals:

- Provisioning
- Registration
- Adding Observers
- Accepting Calls

Provisioning

CiscoMediaTerminals are analogous to physical terminals and must be provisioned accordingly in Cisco CallManager, even though they do not represent actual hardware IP telephones or gateways. Just as IP telephones must be added to Cisco CallManager database using the Device Wizard, CiscoMediaTerminals are added the same way so that Cisco CallManager can associate the application's endpoint with a directory number and other call control properties such as call forwarding. There is no device type called “CiscoMediaTerminal” in the DeviceWizard. Instead, Cisco CallManager has one or more device types that support application registration—each of these types are be exposed as a CiscoMediaTerminal through JTAPI. Currently, only the device type “CTI port” is represented as a CiscoMediaTerminal in JTAPI.
The following procedure lists the steps for provisioning a CTI port for use as an application-controlled endpoint.

**Procedure**

**Step 1** Within the Cisco CallManager configuration web pages, add a “CTI port” device from the Device-Phone web page using the Device Wizard. The CTI port's device name will be the name of the corresponding CiscoMediaTerminal in JTAPI.

**Step 2** Add the new CTI port device, using the User-Global Directory web page, to the list of devices controlled by the application using the User web page.

For more information, refer to the *Cisco CallManager Administration Guide*.

**Registration**

After a media termination device has been properly provisioned in Cisco CallManager, the application may obtain a reference to the corresponding CiscoMediaTerminal object by using either the Provider.getTerminal() method or CiscoProvider.getMediaTerminal() method. The difference between the two methods is that the CiscoProvider.getMediaTerminal() method only returns CiscoMediaTerminals, whereas Provider.getTerminal() will return any terminal object associated with the provider, including those representing physical IP telephones.

Use the CiscoMediaTerminal.register() method to notify Cisco CallManager of their intent to terminate RTP streams of certain payload types. The CiscoMediaTermina.register() method takes an IP address, a port number, and an array of CiscoMediaCapability objects that indicate the types of codecs supported by the application as well as codec-specific parameters.

The IP address and port indicate the address where the application desires to receive media streams. The sample code below demonstrates how to register a CiscoMediaTerminal and bind it to a local address, port number 1234:

```java
ciscoMediaTerminal registerTerminal ( Provider provider, String terminalName ) {  
    final int PORT_NUMBER = 1234;  
    try {  
        CiscoMediaTerminal terminal = provider.getTerminal (  
            terminalName );  
```
In order for this sample code to work, the specified provider must be IN_SERVICE. Further, note that this code uses the constant CiscoMediaCapability.G711_64K_30_MILLISECONDS. This is actually a static reference to a CiscoG711MediaCapability object that specifies a 30 millisecond maximum RTP packet size. This and other common media formats are predefined in the CiscoMediaCapability class.

To specify a media payload that is not listed in the CiscoMediaCapability class, there are two options. If the desired payload type is a simple variation of one of the existing subclasses of CiscoMediaCapability, then constructing a new instance of the subclass is all that is needed. For instance, if an application is willing to support G.711 payloads with a 60-millisecond maximum RTP packet size, it can construct the CiscoG711MediaCapability object directly, specifying 60 milliseconds in the constructor.

On the other hand, if there is no existing subclass of CiscoMediaCapability that matches the desired payload type, construct an instance of the CiscoMediaCapability class directly. Note that the only other parameter that may be specified when constructing a CiscoMediaCapability is the maximum packet size, for example, 30-milliseconds. The following code illustrates registering a custom payload capability:

```java
CiscoMediaTerminal registerTerminal ( Provider provider, String terminalName ) {  
    final int PORT_NUMBER = 1234;  
    try {  
        CiscoMediaTerminal terminal = provider.getTerminal (terminalName );  
        CiscoMediaCapability [] caps = new CiscoMediaCapability [1];  
        caps[0] = new CiscoMediaCapability (RTPPayload.G728,  
            30    // maximum packet size, in milliseconds  
        );  
        terminal.register ( InetAddress.getLocalHost (), PORT_NUMBER,  
            caps );  
    }  
    catch ( Exception e ) {  
        return null;  
    }  
}
```
The payload type parameter used for constructing the CiscoMediaCapability object corresponds to the payload field in the RTP header. The RTPPayload interface defines a number of well-known payload types for this purpose.

### Adding Observers

In order to receive events that indicate where and when to transmit and receive RTP data, place a CiscoTerminalObserver on the CiscoMediaTerminal. Note that the CiscoTerminalObserver extends the standard JTAPI TerminalObserver interface without defining any new methods; it is a marker interface that signals the application's interest in receiving RTP events.

**Note**

This is a TerminalObserver, not a CallObserver, and therefore it must be added using the Terminal.addObserver() method, not the Terminal.addCallObserver() method.

Additionally, add a CallControlCallObserver to the Address object associated with the CiscoMediaTerminal. This guarantees that the application will be notified when calls are offered to the CiscoMediaTerminal. Unlike regular IP telephones, which automatically accept any offered call, CiscoMediaTerminals are responsible for accepting, disconnecting (rejecting), or redirecting any call that is offered to it. Since the CallCtlConnOfferedEv is only presented to CallControlCallObservers placed on Address objects, not Terminal objects, it is important that the application places its CallControlCallObserver in the correct place.

**Note**

Be sure to implement the CallControlCallObserver interface, not just the CallObserver interface; the CallCtlConnOfferedEv will not be delivered to observers that implement only the core CallObserver interface.
Accepting Calls

When an inbound call arrives at the CiscoMediaTerminal's address, it must be accepted using the CallControlConnection.accept() method before a terminal connection is created. This is not necessary for outbound calls — the connection will be in the CallControlConnection.ESTABLISHED state as soon as the call has progressed beyond digit recognition. Once the connection has been accepted, answer the ringing terminal connection to start media flow. Assuming that Cisco CallManager is able to match the capabilities that were registered with the capabilities of the calling endpoint, Cisco CallManager sends the Media Flow events so that the application can begin transmitting and receiving RTP data.

Receiving and Responding to Media Flow Events

Whenever a media stream must be created between two endpoints, Cisco CallManager issues start transmission and start reception events to both endpoints. In JTAPI, the CiscoRTPOutputStartedEv and CiscoRTPInputStartedEv events represent the start transmission and start reception events. The CiscoRTPOutputStartedEv.getRTPOutputProperties() method returns a CiscoRTPOutputProperties object, from which the application can determine the destination address of its peer endpoint in the call, as well as the other RTP properties for the stream such as payload type and packet size. Similarly, the CiscoRTPInputStartedEv.getRTPInputProperties() method returns a CiscoRTPInputProperties object which informs the application of the RTP characteristics for the inbound stream.

At any time while media is flowing, the current CiscoRTPOutputProperties and CiscoRTPInputProperties are also available from the CiscoMediaTerminal.getRTPOutputProperties() and CiscoMediaTerminal.getRTPInputProperties() methods as well. These methods throw an exception if the CiscoMediaTerminal is not currently supposed to transmit or receive media.

When Cisco CallManager wants the application to stop sending or receiving media as the result of a call disconnecting or being put on hold, for example, it sends the CiscoRTPOutputStoppedEv and CiscoRTPInputStoppedEv events. These events mean that the current RTP media stream that exists between the two endpoints should be torn down.
Table 1-5 illustrates the dialogue between Cisco CallManager and a JTAPI application when a call is presented to an application-controlled endpoint. The events in the left column are JTAPI events sent to the CallObserver of the application, and the requests in the left column represent methods invoked by the application.

**Table 1-5  Inbound Media Flow Event**

<table>
<thead>
<tr>
<th>JTAPI Event</th>
<th>Direction</th>
<th>Application Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallActiveEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConnCreatedEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConnProceedingEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCtlConnOfferingEv</td>
<td>←</td>
<td>CallControlConnection.accept ()</td>
</tr>
<tr>
<td>CallCtlConnAlertingEv</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>TermConnCreatedEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TermConnRingingEv</td>
<td>←</td>
<td>TerminalConnection.answer ()</td>
</tr>
<tr>
<td>ConnConnectedEv</td>
<td>←</td>
<td>CallControlConnection.disconnect ()</td>
</tr>
<tr>
<td>CallCtlConnEstablishedEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TermConnTalkingEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CiscoRTPOutputStartedEv</td>
<td>←</td>
<td></td>
</tr>
<tr>
<td>CiscoRTPInputStartedEv</td>
<td>←</td>
<td></td>
</tr>
<tr>
<td>CiscoRTPOutputStoppedEv</td>
<td>←</td>
<td></td>
</tr>
<tr>
<td>CiscoRTPInputStoppedEv</td>
<td>←</td>
<td></td>
</tr>
<tr>
<td>TermConnDroppedEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCtlConnDisconnectedEv</td>
<td>←</td>
<td></td>
</tr>
</tbody>
</table>
Note

Only JTAPI events for the local connection are shown here, that is, for the application endpoint. The actual JTAPI metaevent stream will contain events that describe the state of the calling party.

Cisco IP Telephony Solutions RTP Implementation

The Cisco IP Telephony Solutions architecture puts a premium on performance, and thus some of the features of RTP and its often-associated real-time control protocol (RTCP) are not implemented by Cisco IP Telephony Solutions phones and gateways. To ensure its compatibility, applications must consider the following points:

- As RTCP is not supported. Cisco IP Telephony Solutions endpoints will not send RTCP messages, and they will ignore any such messages sent to them.
- Cisco IP Telephony Solutions endpoints do not currently make use of the synchronization source (SSRC) field in the RTP header, but may in the future. Applications must not multiplex RTP streams using the SSRC field, or phones and gateways may not correctly decode and present the media.

Redirect

JTAPI 1.2 specifies that one of the pre-conditions of the CallControlConnection.redirect() method is for the state of the connection to be in either the CallControlConnection.OFFERING or the CallControlConnection.ALERTING state. Cisco JTAPI also allows a connection in the CallControlConnection.ESTABLISHED state to be redirected.

The redirect() method has the following overloaded form in the CiscoConnection interface. It allows applications to specify the behavior desired when there is a failure while redirecting a call, specifying the calling search space or resetting the original called field. Applications choose the desired behavior, by passing one of the following INT parameters in the overloaded redirect method from the CiscoConnection interface:
• Redirect drop on failure—When a call is directed to a busy or an invalid destination, Cisco CallManager can either drop the call if the redirect fails or leave the call at the redirect controller. The JTAPI application then has a chance to take corrective action, such as redirecting the call to another destination. The option for the redirect mode parameter are as follows:
  - CiscoConnection.REDIRECT_DROP_ON_FAILURE
  - CiscoConnection.REDIRECT_NORMAL

• Calling Address search space—Redirect uses the calling search space parameter to indicate which callingSearchSpace is used. Applications can either use the calling party’s search space or the redirect controller’s search space. The parameter options for this are as follows:
  - CiscoConnection.CALLINGADDRESS_SEARCH_SPACE
  - CiscoConnection.ADDRESS_SEARCH_SPACE

• Resetting original called—The called address option parameter is used to reset the original called fields. The options for this are as follows:
  - CiscoConnection.CALLED_ADDRESS_UNCHANGED
  - CiscoConnection.CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION. This option affects the fields when the call arrives at the redirect destination.

For more information, refer to the com.cisco.jtapi.extensions.CiscoConnection documentation.

Routing

Routing in JTAPI requires the configuration of a CTI Route Point on the Cisco CallManager. Multiple calls can be queued to this Route Point, but only a single line can be configured on a CTI Route Point device. JTAPI implementation of adjunct Routing as described in the call center package includes the following:

• Registering route callbacks on Route Addresses
• Creating appropriate handlers in response to the various routing events (routeSelect, routeEnd)
CTI Route Points are devices with the ability to process any number of incoming calls simultaneously on the same line. Calls may be routed using the methods in the javax.telephony.callcenter package, or calls may be accepted, redirected or disconnected using the methods in the javax.telephony.callcontrol package. Each CTI Route Point may be configured with only one line in the Cisco CallManager. For details on how to configure and administer the CTI Route Point, refer to the Cisco CallManager Administration Guide.

Figure 1-4 shows the CTI Route Point configuration.

Cisco Route Session Implementation

When a call comes into the RouteAddress, the implementation starts a Route Session thread and sends the application a RouteEvent. This thread in turn starts a timer thread to time the application's response to a RouteEvent with either a routeSelect() or an endRoute(). If the application responds with a routeSelect (String[] selectedRoutes ), JTAPI verifies that all preconditions are satisfied and then attempts to route the call to the first destination specified in the array. If the destination is a valid and available number, the call is routed and the application gets a RouteUsedEvent followed by a RouteEndEvent. Otherwise, if there is an error in routing (which may be caused by an invalid/busy/unavailable destination), the application gets a ReRouteEvent. JTAPI starts the Timer Thread again before it sends the re-Route Event. As Cisco CallManager does not support re-Routing, if the routing was unsuccessful, the caller will either hear a busy tone or the call
will be dropped. The application can clean up all failure instances and/or send JTAPI an endRoute to clean up the RouteSession. If the application does not respond with an endRoute(), the JTAPI timer once again expires and JTAPI cleans up the Route Session by sending the application a RouteEndEvent().

If the routing timer expires before the application returns with a selectRoute() or an endRoute() method the Cisco CallManager applies same treatment as when a call is made to an unregistered phone (that is play fast busy). If ForwardNoAnswer is configured on the Route Point, the call is immediately forwarded to that number upon the expiration of the timer.

If the application is unable to respond with a valid address to route the call to, the application may choose to call endRoute with an error. The JTAPI specification defines three errors in the RouteSession interface. These are: ERROR_RESOURCE_BUSY, ERROR_RESOURCE_OUT_OF_SERVICE and ERROR_UNKNOWN. If an endRoute is invoked on the RouteSession, the implementation currently accepts() the call at the RouteAddress so the caller may begin to hear ringback. Thereafter if forwarding is configured for the Route Point, the call will be forwarded when the Forwarding Timer expires.

**Select Route Timer**

This timer is configurable via the JTAPI.ini configuration file which has a key called RouteSelectTimeout=5000. The unit is milliseconds. The default value for this timer is set to 5 seconds, however, depending on the needs of the application, this timer can be extended/decreased to improve Route Session cleanup efficiency. This timer should not be unreasonably large. Each Route Session is a thread and represents a call to the Route Point and we want these Route Sessions to be cleaned up. Should an application expect significant delays between receiving the Route Event and responding with a routeSelect/endRoute event, the application would want to appropriately extend this timer.

**Forwarding Timer**

The timer for Forward on No Answer is currently system wide (i.e., it applies to all devices on Cisco CallManager) and can be configured via the Cisco CallManager Service Parameters configuration. The default value for this timer is set to 12 seconds. In future releases a separate timer for CTI Route Points
will be included so that Forwarding for the Route Point takes into effect immediately after the call is accepted by JTAPI (when the application calls an endRoute or if the routing timer expires).

**RouteSession Extension**

CiscoRouteSession is a Cisco Extension to the JTAPI specification. Most importantly this extension exposes the underlying Call object to the Applications. CiscoRouteSession.getCall() returns CiscoCall and this call exposes other Call Model Objects such as the associated Addresses, Connections etc. The extension also defines additional errors for the application. There is a plan to replace these with an overloaded endRoute(error, action) to specify the call disposition. These include error and action codes—for example the application might want to end the Route Session and trigger the default Forward on No Answer to kick in, in which case the implementation will accept() the call.

**Caller Options Summary**

In the absence of a callback or in the scenario where a routeEvent has not been responded to with a RouteSession.routeSelect() or endRoute(), the caller hears nothing until:

- The application can disconnect() or reject() the connection on the Route Point and thereby the caller hears a busy tone.
- The application can accept the call and then the Forward No Answer if configured will kick-in.
- The application can drop the call. The caller holds the receiver with no clue of what happened.

With a callback, if the application chooses to call an endRoute(), then after endRoute() returns, the caller hears a ringback until:

- client calls a disconnect() which would drop the call.
- The client redirects() the call.
- The forward on no answer timer configured via the scm.ini will kick in and would forward the call unless the above two have already kicked in.
- If no forwarding is configured for the Route Point, the caller continues to hear a ringback unless the first two options kick in.
Fault Tolerance When Using Route Points

One way for an application that uses route points to deal with fault tolerance is to have two JTAPI applications connected to two different Cisco CallManagers, each registering a different RouteAddress. For example, Application1 manages RouteAddress1 using CallManager1. Application2 manages RouteAddress2 using CallManager2. In the Cisco CallManager Administration, the ForwardNoAnswer configuration for these CTI Route Points must be administered so they point to each other. In this example, RouteAddress1 would have FNA=RouteAddress2 and RouteAddress2 would have FNA=RouteAddress1. If CallManager1 goes down, calls are forwarded to RouteAddress2 so that Application2 takes over. Furthermore, both applications could be configured to reconnect to their proper Cisco CallManager server when they receive a ProviderShutdown event.

Redundancy

Devices are configured into device pools, and are assigned static Cisco CallManager groups. Devices register with a particular Cisco CallManager server which handles call control signaling. When a Cisco CallManager server fails, the devices failover to the backup Cisco CallManager server in the group. When the primary Cisco CallManager comes back online and it waits until there are no active calls on the device, then re-homes back to the primary Cisco CallManager server. Cisco JTAPI informs the applications of this transition by sending a temporary out-of-service message while registering to the backup Cisco CallManager server.

Cluster Abstraction

The CTIManager is a virtual representation of all the Cisco CallManagers in a cluster. Cisco JTAPI applications communicate with the CTIManager instead of with a specific Cisco CallManagers. The CTIManager also maintains connection between Cisco CallManagers in a cluster. This allows a provider to represent any devices in the cluster under the CTIManager. Figure 1-5 illustrates “Single box configuration with JTAPI, Cisco CallManager and CTIManager in one box”. Figure 1-6 illustrates “Redundant Cisco CallManager and CTIMangers with JTAPI deployed as a separate client”
For more details about the cluster administration and device pool settings, refer to the Cisco CallManager help pages.

Figure 1-5  Single box configuration with JTAPI, Cisco CallManager and CTIManager in one box

JTAPI
CTIManager
Cisco CallManager

Devices registered on Cisco CallManager
Figure 1-6    Redundant Cisco CallManager and CTIManagers with JTAPI deployed as a separate client

Note
In previous releases of Cisco CallManager, applications running on Cisco JTAPI could only control or monitor devices registered under a single Cisco CallManager. If a Cisco CallManager server went down, the connection between the Cisco CallManager server and JTAPI would terminate and the Provider would shutdown. CTIManager service was only introduced with Cisco CallManager Release 3.1.

Cisco CallManager Server Failure

If a Cisco CallManager server fails, the associated devices re-home to the next Cisco CallManager server in the group. This is defined in the prioritized list of Cisco CallManagers in the device pool information configuration for each device.
Therefore, failure of a Cisco CallManager server only results in a partial outage of devices in the cluster. Those devices are available following a successful Cisco CallManager failover and registration with a secondary Cisco CallManager.

A device such as a Cisco IP Phone 7960 fails over to a secondary Cisco CallManager server only when there are no calls active on that device. The failure of a Cisco CallManager server during a call results only in termination of observation of that device. The media path continues to exist, but without any further call control features.

Cisco JTAPI communicates this partial outage to applications using CiscoAddrOutOfServiceEv and CiscoTermOutOfServiceEv events. When the Cisco CallManager fails over, the device must successfully register to the secondary Cisco CallManager before the device is available to the JTAPI applications. Cisco JTAPI will send the CiscoAddrInServiceEv and CiscoTermInServiceEv events.

The Provider remains in service during this time. Devices on other Cisco CallManager servers are available for call control. The events are sent on callbacks of the respective Address or Terminal observer objects. CiscoAddrOutOfServiceEv and CiscoAddrInServiceEv events are sent to an object implementing the AddressObserver and added to an Address using the addressChangedEvent( ) callback object method. The CiscoTermOutOfServiceEv and CiscoTermInServiceEv events are sent to an object implementing the TerminalObserver interface and added to a Terminal using the terminalChangedEvent( ) call back method.

If the devices are currently in a call, a CallObservationEnded message is sent on the CallObserver callChangedEvent() callback, followed by the CiscoAddrOutOfServiceEv and CiscoTermOutOfServiceEv messages.

Applications must monitor for and respond to the CiscoAddrOutOfServiceEv, CiscoTermOutOfServiceEv, CiscoAddrInServiceEv, and CiscoTermInServiceEv events before the calling call control functions on the Address or Terminal. Applications that do not support this may encounter unexpected errors because the applications are not aware of the exact state of the system.
Redundancy in CTIManagers

Cisco JTAPI also offers applications transparent redundancy via the CTIManager. When the primary CTIManager fails, Cisco JTAPI automatically connects to the backup CTIManager and communicates the reconnection to applications. Instead of connecting to a single Cisco CallManager server, applications now connect to a set of CTIManagers. The CTIManager server names are supplied by the applications when invoking JTAPI.

Cisco JTAPI and the CTIManager maintain bidirectional heartbeat signals to detect a loss of connectivity between them. The CTIManager detects when an application is no longer running and cleans up its allocated resources. Figure 1-7 illustrates the “Logical representation of JTAPI, CTIManager and Cisco CallManager in a cluster”.

From Cisco CallManager Release 3.0 onwards should stop interpreting the out-of-service state as a dead CTI connection. After Cisco JTAPI successfully connects to the primary CTIManager, it will alternately attempt to reconnect to the primary or backup CTIManager if the JTAPI connection to the CTIManager fails.
Invoking CTI Manager Redundancy

When getProvider() method on the CiscoJtapiPeer is called during the application startup, Cisco JTAPI attempts a connection to the first CTIManager in the list and tries a connection to the next CTIManager if connection attempt fails with the first. If all the CTIManagers in the list are not available or if connection is refused by all the CTIManagers, an exception is sent to the application and there are no further reconnection attempts. After the first successful connection, Cisco JTAPI attempts to connect to the backup or primary CTIManager alternatively when a failure to CTIManager or connection to CTIManager is detected.

The list of redundant CTIManagers is a comma-separated list passed into the CiscoJtapiPeer.getProvider(String providerString) method as a String. The usage for the providerString is as follows:

- providerString = CTIManager;login=XXX;passwd=YYY;appinfo=ZZZ  
  (Non-redundant feature)
- providerString = CTIManager1,CTIManager2;login=XXX,passwd=YYY;appinfo=ZZZ  
  (Redundant feature)

Note
The appinfo parameter is optional. If no specific appinfo parameter is provided by the application, Cisco JTAPI generates one from a JTAPI instance ID and the local host name.

Additionally, different CTIManager lists may be defined in the jtapi.ini file to support the CiscoJtapiPeer.getServices() method. Cisco JTAPI accepts the following definition:

CtiManagers=<CTIManager1>,<CTIManager2>;<CTIManager3>

where
<CTIManager1>,<CTIManager2> are a redundant group.
<CTIManager3> is a non redundant group.
CTI Manager Failure

When Cisco JTAPI detects a loss of connection to a CTIManager, the application is notified of this loss in service. The following events are sent to the application on the appropriate Observers:

- A CallObservationEndedEv event is sent to all Call Observers on an Address and calls in progress are ended. The calls are physically connected, but the applications’ observation of the call ends since Cisco JTAPI does not have the ability to send call state changes.

- A CiscoAddrOutOfServiceEv event is sent to all Addresses on a Terminal and a CiscoTermOutOfServiceEv event is sent to the Terminal.

- This process is repeated for all the Terminals in the Provider user controlled list. (A CiscoAddrOutOfServiceEv event is sent only to those Addresses that have an active AddressObserver and a CiscoTermOutOfServiceEv event is sent only to those Terminals that have an active TerminalObserver)

- The Provider is then set in the out-of-service state and the ProvOutOfServiceEv event is delivered on any ProviderObserver callbacks present on the Provider.

Cisco JTAPI then attempts a connection to the next CTIManager in the list and the ProvInServiceEv is sent to the ProviderObserver. The devices previously registered under the application control are reinstated in the new CTIManager. Once the device is reinstated, CiscoAddrInServiceEv and CiscoTermInServiceEv events are sent to the application via the respective Observers. All previously added Observers, are maintained. If there are any existing calls on the devices, a snapshot of the call is sent to the respective CallObservers.

CTIPorts that were previously registered are re-registered with the same media parameters. RouteAddress call backs are maintained as before and these are recovered on the new CTIManager. No call snapshot is however delivered to the RouteAddresses.
Heartbeats

Cisco JTAPI and the CTIManager maintain heartbeat signals to discover a failure in either the CTIManager or JTAPI. The heartbeat is bidirectional with the CTIManager server controlling the heartbeat parameters. Applications can request a desired server heartbeat interval when initializing Cisco JTAPI, but the CTIManager can override it.

Applications specify the desired heartbeat parameter using DesiredServerHeartbeatInterval in the jtapi.ini setting.

Cisco JTAPI specifies the client’s desired heartbeat interval during initialization. The CTIManager specifies to Cisco JTAPI the client side heartbeat interval and the interval at which the server (CTIManager) will send heartbeats. A failure to receive heartbeat message over twice the server-specified interval results in a client-initiated teardown of the connection. To minimize heartbeat traffic, any messages from the client to the server or events from the server to the client substitutes for a heartbeat.

Display Name Interface

The CiscoCall interface is extended to provide methods to get name displays of the calling party and the called party in a call. Applications can use getCurrentCallingPartyDisplayName() to get the display name of the calling party.

JTAPI applications can use the following interface to get the display names of the calling party and the called party.

```java
/**
 * This interface returns the display name of the called party in the call.
 * It returns null if display name is unknown.
 */
public String getCurrentCalledPartyDisplayName();
```

/**
 * This interface returns the display name of the calling party.
 */
public String getCurrentCallingPartyDisplayName();
}

The display name is stored internally in the address objects and updated when currentCallingAddress and currentCalledAddress are updated. NULL is returned if the call is not in the active state and if currentCalling and currentCalled addresses of the call are not initialized.

Note
Call.getCurrentCalledAddress() and call.getCurrentCallingAddress() are not supported for conference calls. Also, call.getCurrentCalledPartyDisplayName() and call.getCurrentCallingPartyDisplayName() are not supported for a conference call.

SetMessageWaiting Interface

SetMessageWaiting provides a method for applications the ability to set the message waiting lamp or indicator for an address. The method is invoked on an Address that is in the same partition as the destination.

The following interface specifies whether the message waiting indicator should be activated or deactivated for the address specified by the destination. If enable is true, message waiting is activated if not already activated. If enable is false, message waiting is deactivated if not already deactivated.

```java
public void setMessageWaiting (java.lang.String destination, boolean enable)
    throws javax.telephony.MethodNotSupportedException,
           javax.telephony.InvalidStateException,
           javax.telephony.PrivilegeViolationException
```
Quite Clear

Quite Clear occurs at the other end when two parties are on a call and one address goes OutOfService because of a network outage, the Cisco CallManager goes down, application controlling CTIPort goes down, or CTIManager goes down. At this stage, the other end of the call can only drop the call or disconnect the connection. It cannot perform any other callControl operations.

For the party that went Out Of Service, applications will see ConnDisconnected/TermConnDroppedEvs and the other end of the call receives ConnFailedEv with CiscoCause of CiscoCallEv.CAUSE_TEMPORARYFAILURE.

If applications try to invoke the following features during Quite Clear mode, PlatformException with error code of CiscoJtapiException.CTIERR_OPERATION_FAILER_QUIETCLEAR is thrown.

- Consult transfer
- Consult conference
- Blind transfer
- Hold
- Unhold

Note

Applications may only drop the call in this mode.

GetCallInfo Interface on Address

GetCallInfo interface on address provides applications the ability to query CallInfo on an address. A query returns the CiscoAddressCallInfo object, which contains information about the number of active or held calls, maximum number of active or held calls, and the Call object for current calls on the address. This interface also provides what calls are at a specific address at a specific time.

Use the following interface to get information of calls present at the terminal:

```java
public CiscoAddressCallInfo getAddressCallInfo(Terminal iterminal);
```
DeleteCall Interface

DeleteCall interface provides applications the ability to delete a call created using the createCall interface. This method accepts a call and throws an InvalidStateException if a provider is not in service or if the call is not in the IDLE state. DeleteCall moves the call to the INVALID state.

The following interface is added toCiscoProvider:

```java
public void deleteCall(Call call) throws InvalidStateException;
```

Applications can use this interface to delete the call created using createCall interface. This method accepts a call and throws an InvalidStateException if the provider is not in service or if the call is not in the IDLE state. DeleteCall moves the call to the INVALID state.

To successfully delete a call, the application creates the call using createCall and the call should be in the IDLE state.

GetGlobalCallID

GetGlobalCallID provides an interface on the CiscoCallID to get the nodeID and the Global Call ID (GCID )of the call; exposing the GCID information available in the internal call object.

The following methods are added to the CiscoCallID interface:

```java
/**
   * returns the callmanager nodeID of the call
   */
public int getCallManagerID();

/**
   * returns the GlobalCallID of the call
   */
public int getGlobalCallID();
```
GetCallID in RTP Events

GetCallID provides an interface on RTP events to access any call information, such as calling party or called party, so that applications can link RTP events with the calls.

The callLegID received in the RTP events from CTIManager is used to determine the ICCNCall on the client side. This call is passed on to the JTAPI layer and the CiscoCall is determined from which CiscoCallID is obtained. This is used to construct the RTP events that are delivered to the application.

The following interface is added to CiscoRTPInputStoppedEv, CiscoRTPInputStartedEv, CiscoRTPOutputStoppedEv, and CiscoRTPOutputStartedEv:

```java
public CiscoCallID getCallID();
```

XSI Object Pass Through

Applications can pass XML objects through JTAPI and CTI interfaces to the phone. The XML object can contain display updates, softkey update/enable/disable and other types of updates on the phone that are available through XSI features. This allows applications to access XSI capabilities through JTAPI and CTI interfaces without maintaining independent connections to the phones.
Chapter 1      Overview

XSI Object Pass Through
Cisco JTAPI Implementation describes interfaces and classes available. To create new applications, use these interfaces and classes with the standard JTAPI interfaces and classes described in the Cisco JTAPI v 1.2.

This chapter contains the following sections:
- Cisco JTAPI Extensions Hierarchy
- Class com.cisco.jtapi.extensions
- Class com.cisco.services.alarm
- Class com.cisco.services.tracing

Cisco JTAPI Extensions Hierarchy

Class Hierarchy

class java.lang.Object
  class com.cisco.services.alarm.AlarmManager
  class com.cisco.services.tracing.BaseTraceWriter (implements com.cisco.services.tracing.TraceWriter)
    class com.cisco.services.tracing.ConsoleTraceWriter
    class com.cisco.services.tracing.LogFileTraceWriter
    class com.cisco.services.tracing.OutputStreamTraceWriter
class com.cisco.services.tracing.SyslogTraceWriter
class com.cisco.jtapi.extensions.CiscoAddressCallInfo
class com.cisco.jtapi.extensions.CiscoJtapiVersion
class com.cisco.jtapi.extensions.CiscoMediaCapability
    class com.cisco.jtapi.extensions.CiscoG711MediaCapability
class com.cisco.jtapi.extensions.CiscoG723MediaCapability
class com.cisco.jtapi.extensions.CiscoG729MediaCapability
class com.cisco.jtapi.extensions.CiscoGSMMediaCapability
class com.cisco.jtapi.extensions.CiscoRTPParams
class com.cisco.services.alarm.DefaultAlarm (implements com.cisco.services.alarm.Alarm)
class com.cisco.services.alarm.DefaultAlarmWriter (implements com.cisco.services.alarm.AlarmWriter)
class com.cisco.services.alarm.ParameterList
class java.lang.Throwable (implements java.io.Serializable)
    class com.cisco.jtapi.extensions.CiscoRegistrationException
    class com.cisco.jtapi.extensions.CiscoUnregistrationException
class com.cisco.services.tracing.TraceManagerFactory

Interface Hierarchy

interface javax.telephony.Address
    interface com.cisco.jtapi.extensions.CiscoAddress (also extends com.cisco.jtapi.extensions.CiscoObjectContainer)
interface javax.telephony.callcenter.RouteAddress
    interface com.cisco.jtapi.extensions.CiscoRouteAddress
interface javax.telephony.AddressObserver
    interface com.cisco.jtapi.extensions.CiscoAddressObserver
interface com.cisco.services.alarm.Alarm
Chapter 2  Cisco JTAPI Implementation

Cisco JTAPI Extensions Hierarchy

interface com.cisco.services.alarm.AlarmWriter

interface javax.telephony.Call
  interface javax.telephony.callcontrol.CallControlCall
    interface com.cisco.jtapi.extensions.CiscoCall (also extends
    com.cisco.jtapi.extensions.CiscoObjectContainer)
  interface com.cisco.jtapi.extensions.CiscoConsultCall

interface com.cisco.jtapi.extensions.CiscoJtapiPeer
  interface com.cisco.jtapi.extensions.CiscoJtapiProperties

interface com.cisco.jtapi.extensions.CiscoObjectContainer
  interface com.cisco.jtapi.extensions.CiscoAddress (also extends
  javax.telephony.Address)

interface com.cisco.jtapi.extensions.CiscoCall (also extends
  javax.telephony.callcontrol.CallControlCall)

interface com.cisco.jtapi.extensions.CiscoConsultCall

interface com.cisco.jtapi.extensions.CiscoCallID

interface com.cisco.jtapi.extensions.CiscoConnection (also extends
  javax.telephony.callcontrol.CallControlConnection)

interface com.cisco.jtapi.extensions.CiscoConnectionID

interface com.cisco.jtapi.extensions.CiscoJtapiPeer (also extends
  javax.telephony.JtapiPeer, com.cisco.services.tracing.TraceModule)

interface com.cisco.jtapi.extensions.CiscoProvider (also extends
  javax.telephony.Provider)

interface com.cisco.jtapi.extensions.CiscoTerminal (also extends
  javax.telephony.Terminal)

interface com.cisco.jtapi.extensions.CiscoMediaTerminal

interface com.cisco.jtapi.extensions.CiscoTerminalConnection (also extends
  javax.telephony.callcontrol.CallControlTerminalConnection)

interface com.cisco.jtapi.extensions.CiscoProvFeatureID

interface com.cisco.jtapi.extensions.CiscoRTPBitRate

interface com.cisco.jtapi.extensions.CiscoRTPInputProperties

interface com.cisco.jtapi.extensions.CiscoRTPOutputProperties
interface com.cisco.jtapi.extensions.CiscoRTPPayload
interface com.cisco.jtapi.extensions.CiscoSynchronousObserver
interface javax.telephony.Connection

interface javax.telephony.callcontrol.CallControlConnection
    interface com.cisco.jtapi.extensions.CiscoConnection (also extends com.cisco.jtapi.extensions.CiscoObjectContainer)

interface javax.telephony.events.Ev

interface javax.telephony.events.AddrEv
    interface com.cisco.jtapi.extensions.CiscoAddrInServiceEv

interface javax.telephony.events.CallEv

interface javax.telephony.events.CallActiveEv
    interface com.cisco.jtapi.extensions.CiscoConsultCallActiveEv (also extends com.cisco.jtapi.extensions.CiscoCallEv)

interface com.cisco.jtapi.extensions.CiscoCallEv (also extends com.cisco.jtapi.extensions.CiscoEv)
    interface com.cisco.jtapi.extensions.CiscoConferenceEndEv
    interface com.cisco.jtapi.extensions.CiscoConferenceStartEv
    interface com.cisco.jtapi.extensions.CiscoConsultCallActiveEv (also extends javax.telephony.events.CallActiveEv)

interface com.cisco.jtapi.extensions.CiscoTransferEndEv
interface com.cisco.jtapi.extensions.CiscoTransferStartEv

interface com.cisco.jtapi.extensions.CiscoEv
    interface com.cisco.jtapi.extensions.CiscoAddrEv (also extends javax.telephony.events.AddrEv)
    interface com.cisco.jtapi.extensions.CiscoAddrInServiceEv
interface com.cisco.jtapi.extensions.CiscoCallEv (also extends javax.telephony.events.CallEv)
interface com.cisco.jtapi.extensions.CiscoConferenceEndEv
interface com.cisco.jtapi.extensions.CiscoConferenceStartEv
interface com.cisco.jtapi.extensions.CiscoConsultCallActiveEv (also extends javax.telephony.events.CallActiveEv)
interface com.cisco.jtapi.extensions.CiscoTransferEndEv
interface com.cisco.jtapi.extensions.CiscoTransferStartEv
interface com.cisco.jtapi.extensions.CiscoOutOfServiceEv
interface com.cisco.jtapi.extensions.CiscoTermOutOfServiceEv (also extends com.cisco.jtapi.extensions.CiscoTermEv)
interface com.cisco.jtapi.extensions.CiscoProvEv (also extends javax.telephony.events.ProvEv)
interface com.cisco.jtapi.extensions.CiscoAddrCreatedEv
interface com.cisco.jtapi.extensions.CiscoAddrRemovedEv
interface com.cisco.jtapi.extensions.CiscoProvFeatureEv
interface com.cisco.jtapi.extensions.CiscoProvCallParkEv
interface com.cisco.jtapi.extensions.CiscoProvFeatureUnRegisteredEv
interface com.cisco.jtapi.extensions.CiscoTermCreatedEv
interface com.cisco.jtapi.extensions.CiscoTermRemovedEv
interface com.cisco.jtapi.extensions.CiscoTermEv (also extends javax.telephony.events.TermEv)
interface com.cisco.jtapi.extensions.CiscoRTPInputStartedEv
interface com.cisco.jtapi.extensions.CiscoRTPInputStoppedEv
interface com.cisco.jtapi.extensions.CiscoRTPOutputStartedEv
interface com.cisco.jtapi.extensions.CiscoRTPOutputStoppedEv
interface com.cisco.jtapi.extensions.CiscoTermDataEv
interface com.cisco.jtapi.extensions.CiscoTermInServiceEv
interface com.cisco.jtapi.extensions.CiscoTermOutOfServiceEv
(also extends com.cisco.jtapi.extensions.CiscoOutOfServiceEv)

interface javax.telephony.events.ProvEv
interface com.cisco.jtapi.extensions.CiscoProvEv (also extends
com.cisco.jtapi.extensions.CiscoEv)
interface com.cisco.jtapi.extensions.CiscoAddrCreatedEv
interface com.cisco.jtapi.extensions.CiscoAddrRemovedEv
interface com.cisco.jtapi.extensions.CiscoProvFeatureEv
interface com.cisco.jtapi.extensions.CiscoProvCallParkEv
interface com.cisco.jtapi.extensions.CiscoProvFeatureUnRegisteredEv
interface com.cisco.jtapi.extensions.CiscoTermCreatedEv
interface com.cisco.jtapi.extensions.CiscoTermRemovedEv

interface javax.telephony.events.TermEv
interface com.cisco.jtapi.extensions.CiscoTermEv (also extends
com.cisco.jtapi.extensions.CiscoEv)
interface com.cisco.jtapi.extensions.CiscoRTPInputStartedEv
interface com.cisco.jtapi.extensions.CiscoRTPInputStoppedEv
interface com.cisco.jtapi.extensions.CiscoRTPOutputStartedEv
interface com.cisco.jtapi.extensions.CiscoRTPOutputStoppedEv
interface com.cisco.jtapi.extensions.CiscoTermDataEv
interface com.cisco.jtapi.extensions.CiscoTermInServiceEv
interface com.cisco.jtapi.extensions.CiscoTermOutOfServiceEv
(also extends com.cisco.jtapi.extensions.CiscoOutOfServiceEv)

interface javax.telephony.JtapiPeer
interface com.cisco.jtapi.extensions.CiscoJtapiPeer (also extends
com.cisco.jtapi.extensions.CiscoObjectContainer,
com.cisco.services.tracing.TraceModule)

interface javax.telephony.Provider
interface com.cisco.jtapi.extensions.CiscoProvider (also extends
com.cisco.jtapi.extensions.CiscoObjectContainer)
interface javax.telephony.capabilities.ProviderCapabilities
  interface com.cisco.jtapi.extensions.CiscoProviderCapabilities
interface javax.telephony.ProviderObserver
  interface com.cisco.jtapi.extensions.CiscoProviderObserver
interface javax.telephony.callcenter.RouteSession
  interface com.cisco.jtapi.extensions.CiscoRouteSession
interface javax.telephony.Terminal
  interface com.cisco.jtapi.extensions.CiscoTerminal (also extends
    com.cisco.jtapi.extensions.CiscoObjectContainer)
  interface com.cisco.jtapi.extensions.CiscoMediaTerminal
interface javax.telephony.TerminalConnection
  interface javax.telephony.callcontrol.CallControlTerminalConnection
    interface com.cisco.jtapi.extensions.CiscoTerminalConnection (also
      extends com.cisco.jtapi.extensions.CiscoObjectContainer)
interface javax.telephony.TerminalObserver
  interface com.cisco.jtapi.extensions.CiscoTerminalObserver
interface com.cisco.services.tracing.Trace
  interface com.cisco.services.tracing.ConditionalTrace
  interface com.cisco.services.tracing.UnconditionalTrace
interface com.cisco.services.tracing.TraceManager
interface com.cisco.services.tracing.TraceModule
  interface com.cisco.jtapi.extensions.CiscoJtapiPeer (also extends
interface com.cisco.services.tracing.TraceWriter
  interface com.cisco.services.tracing.TraceWriterManager
Class `com.cisco.jtapi.extensions`

The Cisco JTAPI extension consists of a set of classes and interfaces that expose the functionality available in the Cisco IP Telephony. This API allows programmers to create independent applications for Cisco CallManager. The Cisco JTAPI implementation offers additional functionality not readily exposed through the JTAPI 1.2 interfaces. Applications can use the interfaces and classes in the `com.cisco.jtapi.extensions` package with the standard JTAPI interfaces and classes described in the Cisco JTAPI v 1.2 Specification manual to create new applications.

### CiscoAddrCreatedEv

#### Declaration

```java
class CiscoAddrCreatedEv extends CiscoProvEv
```

#### All Superinterfaces

- `CiscoEv`, `CiscoProvEv`, `javax.telephony.events.Ev`, `javax.telephony.events.ProvEv`

#### Description

The `CiscoAddrCreatedEv` event

#### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

Member Summary

getAddress()
javax.telephony.Address

Inherited Member Summary

Fields inherited from interface Ev
CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev
getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface ProvEv
getProvider()

Fields

ID
public static final int ID

Methods

getAddress()
public javax.telephony.Address getAddress()
CiscoAddress

Declaration

```java
public interface CiscoAddress extends javax.telephony.Address,
CiscoObjectContainer
```

All Superinterfaces

javax.telephony.Address, CiscoObjectContainer

Description

The `CiscoAddress` interface extends the `Address` interface with additional CallManager-specific capabilities.

See Also

gjavax.telephony.Address

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int EXTERNAL</td>
<td>This is an external address with a valid name.</td>
</tr>
<tr>
<td>static int EXTERNAL_UNKNOWN</td>
<td>This is an external address with an unknown name.</td>
</tr>
<tr>
<td>static int IN_SERVICE</td>
<td>The address is out-of-service</td>
</tr>
<tr>
<td>static int INTERNAL</td>
<td>This is an internal address.</td>
</tr>
<tr>
<td>static int OUT_OF_SERVICE</td>
<td>The address is in-service</td>
</tr>
<tr>
<td>static int RINGER_DEFAULT</td>
<td>Sets the ringer status to configured value</td>
</tr>
</tbody>
</table>
Member Summary

static int RINGER_DISABLE
    Enables the ringer for the address

static int RINGER_ENABLE
    Disables the ringer for the address

static int UNKNOWN
    This is an external address with an unknown name.

Methods

void clearCallConnections()
    Use this interface to clear off any phantom calls on the address

CiscoAddressCallInfo getAddressCallInfo(javax.telephony.Terminal iterminal)
    Use this Interface to get info of calls present at the terminal

int getRegistrationState()
    Returns the state of this address.

int getState()
    Returns the state of this address.

int getType()
    Returns the type of this address.

void setMessageWaiting(java.lang.String destination, boolean enable)
    Specifies whether the message-waiting indicator should be activated or deactivated for
    the Address specified by the destination.

void setRingerStatus(int status)
    Changes the ringer status on this address

Inherited Member Summary

Methods inherited from interface Address
addCallObserver(CallObserver), addObserver(AddressObserver),
getAddressCapabilities(Terminal), getCallObservers(), getCapabilities(),
getConnections(), getName(), getObservers(), getProvider(), getTerminals(),
removeCallObserver(CallObserver), removeObserver(AddressObserver)

Methods inherited from interface CiscoObjectContainer
getAddress(), setObject(Object)
### Fields

**EXTERNAL**

```java
public static final int EXTERNAL
```

This is an external address with a valid name. An address of this type is created when ANI or callerID is available on the call.

**EXTERNAL_UNKNOWN**

```java
public static final int EXTERNAL_UNKNOWN
```

This is an external address with an unknown name. An address of this type is created to represent an endpoint for which there is insufficient information.

**IN_SERVICE**

```java
public static final int IN_SERVICE
```

The address is out-of-service

**INTERNAL**

```java
public static final int INTERNAL
```

This is an internal address.

**OUT_OF_SERVICE**

```java
public static final int OUT_OF_SERVICE
```

The address is in-service

**RINGER_DEFAULT**

```java
public static final int RINGER_DEFAULT
```

Sets the ringer status to configured value

**RINGER_DISABLE**

```java
public static final int RINGER_DISABLE
```

Disables the ringer for the address

**RINGER_ENABLE**

```java
public static final int RINGER_ENABLE
```
Enables the ringer for the address

**UNKNOWN**

```java
public static final int UNKNOWN
```

This is an external address with an unknown name. An address of this type is
created to represent an endpoint for which there is insufficient information.

## Methods

### clearCallConnections()

```java
public void clearCallConnections()
```

Use this interface to clear off any phantom calls on the address

**Throws:**

`javax.telephony.PrivilegeViolationException`

### getAddressCallInfo(Terminal)

```java
public com.cisco.jtapi.extensions.CiscoAddressCallInfo
    getAddressCallInfo(javax.telephony.Terminal
terminal)
```

Use this Interface to get info of calls present at the terminal

### getRegistrationState()

```java
public int getRegistrationState()
```

**Deprecated.** This method has been replaced by the getState() method.

Returns the state of this address.

The state may be any of the following constants:

- `CiscoAddress.OUT_OF_SERVICE`
- `CiscoAddress.IN_SERVICE`

**Returns:** the state of this address

### getState()

```java
public int getState()
```
Returns the state of this address.

The state may be any of the following constants:

- `CiscoAddress.OUT_OF_SERVICE`
- `CiscoAddress.IN_SERVICE`

**Returns:** the state of this address

```java
getType()
```

public int `getType()`

Returns the type of this address.

The type may be any of the following constants:

- `CiscoAddress.INTERNAL`
- `CiscoAddress.EXTERNAL`
- `CiscoAddress.EXTERNAL_UNKNOWN`

**Returns:** the type of address

```java
setMessageWaiting(String, boolean)
```

public void `setMessageWaiting(java.lang.String destination, boolean enable)`

Throws `MethodNotSupportedException`, `InvalidStateException`, `PrivilegeViolationException`

Specifies whether the message-waiting indicator should be activated or deactivated for the Address specified by the `destination`. If `enable` is true, message-waiting is activated if not already activated. If `enable` is false, message-waiting is deactivated if not already deactivated.

**Pre-conditions:**

1. `(this.getProvider()).getState() == Provider.IN_SERVICE`

**Post-conditions:**

1. `(this.getProvider()).getState() == Provider.IN_SERVICE`

**Note:** The following postconditions as specified in `CallControlAddress` are currently not enforced by this implementation.

1. `this.getMessageWaiting() == enable`
2. CallCtlAddrMessageWaitingEv is delivered for this Address

Parameters:
- destination - DN whose message waiting indicator should be activated
- enable - True to activate message-waiting, false to deactivate.

Throws:
- javax.telephony.MethodNotSupportedException - This method is not supported by the given implementation.
- javax.telephony.InvalidStateException - The Provider is not “in service”.
- javax.telephony.PrivilegeViolationException - The Provider user has insufficient privileges to invoke the message waiting indicator for this destination

setRingerStatus(int)

public void setRingerStatus(int status)
        throws MethodNotSupportedException, InvalidStateException, InvalidArgumentException

changes the ringer status on this address

Accepts on of the following constants: CiscoAddress.RINGER_DEFAULT CiscoAddress.RINGER_DISABLE CiscoAddress.RINGER_ENABLE

Throws:
- javax.telephony.InvalidArgumentException
- javax.telephony.InvalidStateException
- javax.telephony.MethodNotSupportedException

CiscoAddressCallInfo

Declaration

public class CiscoAddressCallInfo
java.lang.Object
   +--com.cisco.jtapi.extensions.CiscoAddressCallInfo

### Member Summary

#### Constructors

public `CiscoAddressCallInfo` (int inumActiveCalls, int imaxActiveCalls, int inumCallsOnHold, int imaxCallsOnHold)

public `CiscoAddressCallInfo` (int inumActiveCalls, int imaxActiveCalls, int inumCallsOnHold, int imaxCallsOnHold, CiscoCall icalls)

#### Methods

- `getCalls()`: Returns an array of `CiscoCall` objects.
- `getMaxActiveCalls()`: Returns the maximum number of active calls.
- `getMaxCallsOnHold()`: Returns the maximum number of calls on hold.
- `getNumActiveCalls()`: Returns the number of active calls.
- `getNumCallsOnHold()`: Returns the number of calls on hold.

### Inherited Member Summary

Methods inherited from class `Object`

- `clone()`, `equals(Object)`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `toString()`, `wait()`, `wait()`, `wait()`
Chapter 2  Cisco JTAPI Implementation

CiscoAddressObserver

**CiscoAddressCallInfo**(int, int, int, int, CiscoCall[]):

```java
public CiscoAddressCallInfo(int inumActiveCalls, 
  int imaxActiveCalls, int inumCallsOnHold, 
  int imaxCallsOnHold, 
  com.cisco.jtapi.extensions.CiscoCall[] icalls)
```

Methods

- getCalls()
  ```java
  public com.cisco.jtapi.extensions.CiscoCall[] getCalls()
  ```
- getMaxActiveCalls()
  ```java
  public int getMaxActiveCalls()
  ```
- getMaxCallsOnHold()
  ```java
  public int getMaxCallsOnHold()
  ```
- getNumActiveCalls()
  ```java
  public int getNumActiveCalls()
  ```
- getNumCallsOnHold()
  ```java
  public int getNumCallsOnHold()
  ```

CiscoAddressObserver

**Declaration**

```java
public interface CiscoAddressObserver extends 
  javax.telephony.AddressObserver
```

**All Superinterfaces**

javax.telephony.AddressObserver
CiscoAddrEv

Description

Applications implement this interface in order to receive CiscoAddrEv events such as CiscoAddrInServiceEv and CiscoAddrOutOfServiceEv when observing Addresses via the Address.addObserver method.

See Also:
CiscoAddrInServiceEv, CiscoAddrOutOfServiceEv

Inherited Member Summary

<table>
<thead>
<tr>
<th>Methods inherited from interface AddressObserver</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressChangedEvent(AddrEv[])</td>
</tr>
</tbody>
</table>

CiscoAddrEv

Declaration

public interface CiscoAddrEv extends CiscoEv,
javax.telephony.events.AddrEv

All Superinterfaces

javax.telephony.events.AddrEv, CiscoEv,
javax.telephony.events.Ev

All Known Subinterfaces

CiscoAddrInServiceEv, CiscoAddrOutOfServiceEv
Chapter 2  Cisco JTAPI Implementation

Description

The CiscoAddrEv interface, which extends JTAPI’s core javax.telephony.events.AddrEv interface, serves as the base interface for all Cisco-extended JTAPI Address events. Every Address-related event in this package extends this interface, directly or indirectly.

See Also:
javax.telephony.events.AddrEv

Inherited Member Summary

Fields inherited from interface Ev
CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface AddrEv
getAddress()

Methods inherited from interface Ev
getaCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

CiscoAddrInServiceEv

Declaration

public interface CiscoAddrInServiceEv extends CiscoAddrEv
All Superinterfaces

javax.telephony.events.AddrEv, CiscoAddrEv, CiscoEv, javax.telephony.events.Ev

Description

The CiscoAddrInServiceEv event

Member Summary

Fields

static int ID

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, 
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, 
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, 
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, 
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, 
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface AddrEv

getAddress()
Chapter 2      Cisco JTAPI Implementation

CiscoAddrOutOfServiceEv

Fields

public static final int ID

Declaration

public interface CiscoAddrOutOfServiceEv extends CiscoAddrEv,
CiscoOutOfServiceEv

All Superinterfaces

javax.telephony.events.AddrEv, CiscoAddrEv, CiscoEv,
CiscoOutOfServiceEv, javax.telephony.events.Ev

Description

The CiscoAddrOutOfServiceEv event

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
</tr>
</tbody>
</table>
Inherited Member Summary

Fields inherited from interface CiscoOutOfServiceEv

CAUSE_CALLMANAGER_FAILURE, CAUSE_CTIMANAGER_FAILURE, CAUSE_DEVICE_FAILURE, CAUSE_DEVICE_UNREGISTERED, CAUSE_NOCALLMANAGER_AVAILABLE, CAUSE_REHOME_TO_HIGHER_PRIORITY_CM, CAUSE_REHOMING_FAILURE

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface AddrEv

getAddress()

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Fields

ID

public static final int ID

CiscoAddrRemovedEv

Declaration

public interface CiscoAddrRemovedEv extends CiscoProvEv
All Superinterfaces

`CiscoEv, CiscoProvEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv`

Description

The `CiscoAddrRemovedEv` event

### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td><code>getAddress()</code></td>
</tr>
</tbody>
</table>

### Inherited Member Summary

**Fields inherited from interface `Ev`**

`CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFER, META_SNAPSHOT, META_UNKNOWN`

**Methods inherited from interface `Ev`**

`getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()`

**Methods inherited from interface `ProvEv`**
Inherited Member Summary

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getProvider()</td>
</tr>
</tbody>
</table>

Fields

- public static final int ID

Methods

- public javax.telephony.Address getAddress()

CiscoCall Declaration

```
public interface CiscoCall extends 
    javax.telephony.callcontrol.CallControlCall, 
    CiscoObjectContainer
```

All Superinterfaces

```
javax.telephony.Call, 
javax.telephony.callcontrol.CallControlCall, 
CiscoObjectContainer
```
## All Known Subinterfaces

### CiscoConsultCall

#### Description

The `CiscoCall` interface extends the `CallControlCall` interface with additional CallManager-specific capabilities.

In CallManager terms, every Call object comprises a set of **call legs** that share a common identifier: the **global call handle**. Connection objects represent call legs in JTAPI, and the Call object that relates a set of Connections contains the global call handle that the underlying call legs share.

The global call handle within a `CiscoCall` is accessible via its `CallManagerID` and `CallID` properties. Taken together, the `CallManagerID` and `CallID` form the global call handle maintained by the CallManager. This pair of properties is guaranteed to be unique among all **ACTIVE** Call objects, but when an **ACTIVE** call becomes **INACTIVE**, its `CallManagerID` and `CallID` may be reused to identify a newly-created Call object. Therefore, it is possible for an **INACTIVE** Call to have identical `CallManagerID` and `CallID` properties to those of a currently **ACTIVE** Call object.

#### See Also:

`javax.telephony.Call`

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><code>CiscoCallID</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><code>javax.telephony.Address</code></td>
</tr>
<tr>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Member Summary

javax.telephony.Address

getCurrentCallingAddress()
This interface returns current called address for the call this will return updated calling address every every time call is redirected or transferred

NOTE:
In the CiscoJtapi implementation
CallControlCall.getCallingAddress() returns the first calling party of the call i.e.

java.lang.String

getCurrentCallingPartyDisplayName()
This interface returns the display name of the calling party.

Inherited Member Summary

Fields inherited from interface Call
ACTIVE, IDLE, INVALID

Methods inherited from interface Call
addObserver(CallObserver), connect(Terminal, Address, String),
getCallCapabilities(Terminal, Address), getCapabilities(Terminal, Address),
getConnections(), getObservers(), getProvider(), getState(),
removeObserver(CallObserver)

Methods inherited from interface CallControlCall
addParty(String), conference(Call), consult(TerminalConnection),
consult(TerminalConnection), drop(), getCalledAddress(), getCallingAddress(),
getCallingTerminal(), getConferenceController(), getConferenceEnable(),
getLastRedirectedAddress(), getTransferController(), getTransferEnable(),
offHook(Address, Terminal), setConferenceController(TerminalConnection),
setConferenceEnable(boolean), setTransferController(TerminalConnection),
setTransferEnable(boolean), transfer(String), transfer(String)

Methods inherited from interface CiscoObjectContainer
getObject(), setObject(Object)
## Methods

**getCallID()**

```java
public com.cisco.jtapi.extensions.CiscoCallID getCallID()
```

CallID is a unique identifier among all ACTIVE calls with the same CallManagerID.

**Returns:** the CallID property of this Call

**getCurrentCalledAddress()**

```java
public javax.telephony.Address getCurrentCalledAddress()
```

This interface returns the display of the called party in the call. It returns null if display name is unknown.

**getCurrentCallingAddress()**

```java
public javax.telephony.Address getCurrentCallingAddress()
```

This interface returns current called address for the call this will return updated calling address every every time call is redirected or transferred

**NOTE:**

In the CiscoJtapi implementation `CallControlCall.getCallingAddress()` returns the first calling party of the call i.e. the original calling party

**Usage:**

```java
if ( call instanceof CiscoCall ) {
    Address currentCalled = ((CiscoCall)call).getCurrentCalling ();
}
```

**See Also:**

`javax.telephony.callcontrol.CallControlCall`
getCurrentCallingPartyDisplayName()

public java.lang.String getCurrentCallingPartyDisplayName()

This interface returns the display name of the calling party. It returns null if display name is unknown.

CiscoCallEv

Declaration

public interface CiscoCallEv extends CiscoEv, javax.telephony.events.CallEv

All Superinterfaces

javax.telephony.events.CallEv, CiscoEv, javax.telephony.events.Ev

All Known Subinterfaces

CiscoConferenceEndEv, CiscoConferenceStartEv, CiscoConsultCallActiveEv, CiscoTransferEndEv, CiscoTransferStartEv

Description

The CiscoCallEv interface, which extends JTAPI’s core javax.telephony.events.CallEv interface, serves as the base interface for all Cisco-extended JTAPI Call events. Every Call-related event in this package extends this interface, directly or indirectly.

See Also:

javax.telephony.events.CallEv
### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int CAUSE_ACCESSINFORMATIONDISCARDED</td>
<td>CAUSE_ACCESSINFORMATIONDISCARDED</td>
</tr>
<tr>
<td>static int CAUSE_BCBPRESENTLYAVAIL</td>
<td>CAUSE_BCBPRESENTLYAVAIL</td>
</tr>
<tr>
<td>static int CAUSE_BCNAUTHORIZED</td>
<td>CAUSE_BCNAUTHORIZED</td>
</tr>
<tr>
<td>static int CAUSE_BEARERCAPNIMPL</td>
<td>CAUSE_BEARERCAPNIMPL</td>
</tr>
<tr>
<td>static int CAUSE_CALLBEINGDELIVERED</td>
<td>CAUSE_CALLBEINGDELIVERED</td>
</tr>
<tr>
<td>static int CAUSE_CALLIDINUSE</td>
<td>CAUSE_CALLIDINUSE</td>
</tr>
<tr>
<td>static int CAUSE_CALLMANAGER_FAILURE</td>
<td>CAUSE_CALLMANAGER_FAILURE</td>
</tr>
<tr>
<td>static int CAUSE_CALLREJECTED</td>
<td>CAUSE_CALLREJECTED</td>
</tr>
<tr>
<td>static int CAUSE_CALLSPLIT</td>
<td>CAUSE_CALLSPLIT</td>
</tr>
<tr>
<td>static int CAUSE_CHANYPENIMPL</td>
<td>CAUSE_CHANYPENIMPL</td>
</tr>
<tr>
<td>static int CAUSE_CHANUNACCEPTABLE</td>
<td>CAUSE_CHANUNACCEPTABLE</td>
</tr>
<tr>
<td>static int CAUSE_CTIMANAGER_FAILURE</td>
<td>CAUSE_CTIMANAGER_FAILURE</td>
</tr>
<tr>
<td>static int CAUSE_DESTINATIONOUTOFORDER</td>
<td>CAUSE_DESTINATIONOUTOFORDER</td>
</tr>
<tr>
<td>static int CAUSE_DESTNUMMISSANDDCNOTSUB</td>
<td>CAUSE_DESTNUMMISSANDDCNOTSUB</td>
</tr>
<tr>
<td>static int CAUSE_FACILITYREJECTED</td>
<td>CAUSE_FACILITYREJECTED</td>
</tr>
<tr>
<td>static int CAUSE_IDENTIFIEDCHANDOESNOTEXIST</td>
<td>CAUSE_IDENTIFIEDCHANDOESNOTEXIST</td>
</tr>
<tr>
<td>static int CAUSE_IENIMPL</td>
<td>CAUSE_IENIMPL</td>
</tr>
<tr>
<td>static int CAUSE_INBOUNDBLINDTRANSFER</td>
<td>CAUSE_INBOUNDBLINDTRANSFER</td>
</tr>
<tr>
<td>static int CAUSE_INBOUNDCONFERENCE</td>
<td>CAUSE_INBOUNDCONFERENCE</td>
</tr>
<tr>
<td>static int CAUSE_INBOUNDTRANSFER</td>
<td>CAUSE_INBOUNDTRANSFER</td>
</tr>
<tr>
<td>static int CAUSE_INCOMINGCALLBARRED</td>
<td>CAUSE_INCOMINGCALLBARRED</td>
</tr>
<tr>
<td>static int CAUSE_INCOMPATABLEDESTINATION</td>
<td>CAUSE_INCOMPATABLEDESTINATION</td>
</tr>
<tr>
<td>static int CAUSE_INTERWORKINGUNSPECIFIED</td>
<td>CAUSE_INTERWORKINGUNSPECIFIED</td>
</tr>
<tr>
<td>static int CAUSE_INVALIDCALLREFVALUE</td>
<td>CAUSE_INVALIDCALLREFVALUE</td>
</tr>
<tr>
<td>static int CAUSE_INVALIDIECONTENTS</td>
<td>CAUSE_INVALIDIECONTENTS</td>
</tr>
<tr>
<td>static int CAUSE_INVALIDMESSAGEUNSPECIFIED</td>
<td>CAUSE_INVALIDMESSAGEUNSPECIFIED</td>
</tr>
<tr>
<td>static int CAUSE_INVALIDNUMBERFORMAT</td>
<td>CAUSE_INVALIDNUMBERFORMAT</td>
</tr>
<tr>
<td>static int CAUSE_INVALIDTRANSITNETSEL</td>
<td>CAUSE_INVALIDTRANSITNETSEL</td>
</tr>
<tr>
<td>static int CAUSE_MANDATORYIEMISSING</td>
<td>CAUSE_MANDATORYIEMISSING</td>
</tr>
<tr>
<td>static int CAUSE_MSGCOMPATABLEWCS</td>
<td>CAUSE_MSGCOMPATABLEWCS</td>
</tr>
<tr>
<td>static int CAUSE_MSGTYPENCOMPATWCS</td>
<td>CAUSE_MSGTYPENCOMPATWCS</td>
</tr>
<tr>
<td>static int CAUSE_MSGTYPENIMPL</td>
<td>CAUSE_MSGTYPENIMPL</td>
</tr>
<tr>
<td>static int CAUSE_NETOUTOFORDER</td>
<td>CAUSE_NETOUTOFORDER</td>
</tr>
<tr>
<td>static int CAUSE_NOANSWERFROMUSER</td>
<td>CAUSE_NOANSWERFROMUSER</td>
</tr>
<tr>
<td>static int CAUSE_NOCALLSSUSPENDED</td>
<td>CAUSE_NOCALLSSUSPENDED</td>
</tr>
</tbody>
</table>
Member Summary

static int CAUSE_NOCIRCAVAIL
static int CAUSE_NOERROR
static int CAUSE_NONSELECTEDUSERCLEARING
static int CAUSE_NORMALCALLCLEARING
static int CAUSE_NORMALUNSPECIFIED
static int CAUSE_NOROUTETODestination
static int CAUSE_NOROUTETOTRANsitNET
static int CAUSE_NOUSERRESPONDING
static int CAUSE_NUMBERCHANGED
static int CAUSE_ONLYRDIVEARERCAPAvaIL
static int CAUSE_OUTBOUNDCONFERENCE
static int CAUSE_OUTBOUNDTRANSFER
static int CAUSE_PROTOCOLERRORUNSPECIFIED
static int CAUSE_QUALOFSERVNAVAIL
static int CAUSE_RECOVERYONTIMEREXPIRY
static int CAUSE_REDIRECTED
static int CAUSE_REQCALLIDHASBEENCLEARED
static int CAUSE_REQCIRCNAvaIL
static int CAUSE_REQFACILITYNIMPL
static int CAUSE_REQFACILITYNOTSUBSCRIBED
static int CAUSE_RESOURCESAVAIL
static int CAUSE_RESPONSETOSTATUSENQUIRY
static int CAUSE_SERVNOTAVALUUNSPECIFIED
static int CAUSE_SERVOPERATIONVIOLATED
static int CAUSE_SERVOROPTNAVAILORIMPL
static int CAUSE_SUSPCALLBUTNOTTHISONE
static int CAUSE_SWITCHINGEQUIPMENTCONGESTION
static int CAUSE_TEMPORARYFAILURE
static int CAUSE_UNALLOCATEDNUMBER
static int CAUSE_USERBUSY

Methods

int getCiscoCause()
Returns the CallManager cause for this event.
Inherited Member Summary

Fields inherited from interface `Ev`

CAUSE_ACCESSINFORMATIONDISCARDED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE.Unknown, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface `CallEv`

getCall()

Methods inherited from interface `Ev`

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Fields

public static final int CAUSE_ACCESSINFORMATIONDISCARDED

CAUSE_BCBPRESENTLYAVAIL

public static final int CAUSE_BCBPRESENTLYAVAIL

CAUSE_BCNAUTHORIZED

public static final int CAUSE_BCNAUTHORIZED

CAUSE_BEARERCAPNIMPL

public static final int CAUSE_BEARERCAPNIMPL

CAUSE_CALLBEINGDELEVERED

public static final int CAUSE_CALLBEINGDELEVERED

CAUSE_CALLIDINUSE

public static final int CAUSE_CALLIDINUSE
Chapter 2      Cisco JTAPI Implementation

CAUSE_CALLMANAGER_FAILURE
   public static final int CAUSE_CALLMANAGER_FAILURE

CAUSE_CALLREJECTED
   public static final int CAUSE_CALLREJECTED

CAUSE_CALLSPLIT
   public static final int CAUSE_CALLSPLIT

CAUSE_CHANUNACCEPTABLE
   public static final int CAUSE_CHANUNACCEPTABLE

CAUSE_DESTINATIONOUTOFORDER
   public static final int CAUSE_DESTINATIONOUTOFORDER

CAUSE_DESTNUMMISSANDDCNOTSUB
   public static final int CAUSE_DESTNUMMISSANDDCNOTSUB

CAUSE_FACILITYREJECTED
   public static final int CAUSE_FACILITYREJECTED

CAUSE_IDENTIFIEDCHANDOESNOTEXIST
   public static final int CAUSE_IDENTIFIEDCHANDOESNOTEXIST

CAUSE_IENIMPL
   public static final int CAUSE_IENIMPL

CAUSE_INBOUNDBLINDTRANSFER
   public static final int CAUSE_INBOUNDBLINDTRANSFER

Cisco JTAPI Developer Guide for Cisco CallManager 3.3(2)

OL-3022-01
CAUSE_INBOUNDCONFERENCE
   public static final int CAUSE_INBOUNDCONFERENCE

CAUSE_INBOUNDTRANSFER
   public static final int CAUSE_INBOUNDTRANSFER

CAUSE_INCOMINGCALLBARRED
   public static final int CAUSE_INCOMINGCALLBARRED

CAUSE_INCOMPATABLEDESTINATION
   public static final int CAUSE_INCOMPATABLEDESTINATION

CAUSE_INTERWORKINGUNSPECIFIED
   public static final int CAUSE_INTERWORKINGUNSPECIFIED

CAUSE_INVALIDCALLREFVALUE
   public static final int CAUSE_INVALIDCALLREFVALUE

CAUSE_INVALIDIECONTENTS
   public static final int CAUSE_INVALIDIECONTENTS

CAUSE_INVALIDMESSAGEUNSPECIFIED
   public static final int CAUSE_INVALIDMESSAGEUNSPECIFIED

CAUSE_INVALIDNUMBERFORMAT
   public static final int CAUSE_INVALIDNUMBERFORMAT

CAUSE_INVALIDTRANSITNETSEL
   public static final int CAUSE_INVALIDTRANSITNETSEL

CAUSE_MANDATORYIEMISSING
   public static final int CAUSE_MANDATORYIEMISSING

CAUSE_MSGNCOMPATABLEWCS
   public static final int CAUSE_MSGNCOMPATABLEWCS
<table>
<thead>
<tr>
<th>CAUSE</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGTYPENCOMPATWCS</td>
<td>0x01</td>
<td>Message Type - Network Compatibility with Cisco WCS</td>
</tr>
<tr>
<td>MSGTYPENIMPL</td>
<td>0x02</td>
<td>Message Type - Network Implementation</td>
</tr>
<tr>
<td>NETOUTOFORDER</td>
<td>0x03</td>
<td>Network Out of Order</td>
</tr>
<tr>
<td>NOANSWERFROMUSER</td>
<td>0x04</td>
<td>No Answer from User</td>
</tr>
<tr>
<td>NOCALLSUSPENDED</td>
<td>0x05</td>
<td>Call Suspended</td>
</tr>
<tr>
<td>NOCIRCAVAIL</td>
<td>0x06</td>
<td>No Circuit Available</td>
</tr>
<tr>
<td>NOERROR</td>
<td>0x07</td>
<td>No Error</td>
</tr>
<tr>
<td>NONSELECTEDUSERCLEARING</td>
<td>0x08</td>
<td>Non-Selected User Clearing</td>
</tr>
<tr>
<td>NORMALCALLCLEARING</td>
<td>0x09</td>
<td>Normal Call Clearing</td>
</tr>
<tr>
<td>NORMALUNSPECIFIED</td>
<td>0x0A</td>
<td>Normal Unspecified</td>
</tr>
<tr>
<td>NOROUTETODDESTINATION</td>
<td>0x0B</td>
<td>No Route to Destination</td>
</tr>
<tr>
<td>NOROUTETOTRANSITNET</td>
<td>0x0C</td>
<td>No Route to Transit Network</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

public static final int CAUSE_NOUSERRESPONDING

public static final int CAUSE_NUMBERCHANGED

public static final int CAUSE_ONLYRDIVEARERCAPAVAIL

public static final int CAUSE_OUTBOUNDTRANSFER

public static final int CAUSE_SUITEERRORUNSPECIFIED

public static final int CAUSE_QUALOFSERVNAVAIL

public static final int CAUSE_RECOVERYONTIMEREXPIRY

public static final int CAUSE_REDIRECTED

public static final int CAUSE_REQCIRCNAVIL

public static final int CAUSE_REQFACILITYNIMPL
<table>
<thead>
<tr>
<th>CAUSE</th>
<th>Description</th>
<th>Java Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAUSE_REQFACILITYNOTSUBSCRIBED</td>
<td>Facility requested is not subscribed</td>
<td>CAUSE_REQFACILITYNOTSUBSCRIBED</td>
</tr>
<tr>
<td>CAUSE_RESOURCESNAVAIL</td>
<td>Resources not available</td>
<td>CAUSE_RESOURCESNAVAIL</td>
</tr>
<tr>
<td>CAUSE_RESPONSETOSTATUSENQUIRY</td>
<td>Response to status inquiry</td>
<td>CAUSE_RESPONSETOSTATUSENQUIRY</td>
</tr>
<tr>
<td>CAUSE_SERVNOTAVAILUNSPECIFIED</td>
<td>Service not available unspecified</td>
<td>CAUSE_SERVNOTAVAILUNSPECIFIED</td>
</tr>
<tr>
<td>CAUSE_SERVOPERATIONVIOLATED</td>
<td>Service operation violated</td>
<td>CAUSE_SERVOPERATIONVIOLATED</td>
</tr>
<tr>
<td>CAUSE_SERVOROPTNAVAILORIMPL</td>
<td>Service or option not available or implied</td>
<td>CAUSE_SERVOROPTNAVAILORIMPL</td>
</tr>
<tr>
<td>CAUSE_SUSPCALLBUTNOTTHISONE</td>
<td>Suspicious call but not this one</td>
<td>CAUSE_SUSPCALLBUTNOTTHISONE</td>
</tr>
<tr>
<td>CAUSE_SWITCHINGEQUIPMENTCONGESTION</td>
<td>Switching equipment congestion</td>
<td>CAUSE_SWITCHINGEQUIPMENTCONGESTION</td>
</tr>
<tr>
<td>CAUSE_TEMPORARYFAILURE</td>
<td>Temporary failure</td>
<td>CAUSE_TEMPORARYFAILURE</td>
</tr>
<tr>
<td>CAUSE_UNALLOCATEDNUMBER</td>
<td>Unallocated number</td>
<td>CAUSE_UNALLOCATEDNUMBER</td>
</tr>
<tr>
<td>CAUSE_USERBUSY</td>
<td>User busy</td>
<td>CAUSE_USERBUSY</td>
</tr>
</tbody>
</table>
Methods

getCiscoCause()

public int getCiscoCause()

Returns the CallManager cause for this event.

In order to function properly, some applications need to know the reason why an event happened at an endpoint that the application is observing. For example, a Connection may be disconnected because the call was not answered (CAUSE_NOANSWERFROMUSER), or whether the caller it was disconnected because it was rejected (CAUSE_CALLREJECTED).

Returns: the CallManager cause for this event

CiscoCallID

Declaration

public interface CiscoCallID extends CiscoObjectContainer

All Superinterfaces

CiscoObjectContainer

Description

The CiscoCallID object represents a unique object associated with each call. Applications may use the object itself or the integer representation of the object returned by the intValue() method.
CiscoCallID

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getCall()</strong></td>
</tr>
<tr>
<td>public com.cisco.jtapi.extensions.CiscoCall <strong>getCall()</strong></td>
</tr>
<tr>
<td><strong>getCallManagerID()</strong></td>
</tr>
<tr>
<td>public int <strong>getCallManagerID()</strong></td>
</tr>
<tr>
<td>returns the call manager nodeID of the call</td>
</tr>
<tr>
<td><strong>getGlobalCallID()</strong></td>
</tr>
<tr>
<td>public int <strong>getGlobalCallID()</strong></td>
</tr>
<tr>
<td>returns the GlobalCallID of the call</td>
</tr>
<tr>
<td><strong>intValue()</strong></td>
</tr>
<tr>
<td>public int <strong>intValue()</strong></td>
</tr>
<tr>
<td>Returns an integer representation of this object, currently a bitwise OR of the CallManagerID and GlobalCallID properties (shifted and truncated appropriately)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from interface **CiscoObjectContainer**

**getObject()**, **setObject(Object)**

Methods

- **getCall()**
  ```java
  public com.cisco.jtapi.extensions.CiscoCall getCall()
  ```
- **getCallManagerID()**
  ```java
  public int getCallManagerID()
  ```
  returns the call manager nodeID of the call
- **getGlobalCallID()**
  ```java
  public int getGlobalCallID()
  ```
  returns the GlobalCallID of the call
- **intValue()**
  ```java
  public int intValue()
  ```
Returns an integer representation of this object, currently a bitwise OR of
the CallManagerID and GlobalCallID properties (shifted and truncated
appropriately)

Returns: an integer representation of this object

CiscoConferenceEndEv

Declaration

public interface CiscoConferenceEndEv extends CiscoCallEv

All Superinterfaces

javax.telephony.events.CallEv, CiscoCallEv, CiscoEv,
javax.telephony.events.Ev

Description

The CiscoConferenceEndEv event indicates that a transfer operation has
completed. This event is reported via the CallControlCallObserver
interface.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td>getConferenceControllerAddress()</td>
</tr>
</tbody>
</table>
|                 | Returns the Address which currently acts as the conference controller for this call ——
|                 | the initiating call.                                        |
Member Summary

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.telephony.Call</td>
<td>getConferencedCall()</td>
<td>Returns the call that has been merged.</td>
</tr>
<tr>
<td>javax.telephony.Call</td>
<td>getFinalCall()</td>
<td>Returns the call that remains active after the conference is completed.</td>
</tr>
<tr>
<td>javax.telephony.TerminatorConnection</td>
<td>getHeldConferenceController()</td>
<td>Returns the TerminalConnection which currently acts as the conference controller for this call — the final call.</td>
</tr>
<tr>
<td>javax.telephony.TerminatorConnection</td>
<td>getTalkingConferenceController()</td>
<td>Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Fields inherited from interface CiscoCallEv
### Inherited Member Summary

<table>
<thead>
<tr>
<th>Members inherited from interface Ev</th>
</tr>
</thead>
<tbody>
<tr>
<td>cause_accessinformationdiscarded, cause_bcbpresentlyavail, cause_bcnauthorized, cause_bearercapnimpl, cause_callbeingdelivered, cause_callidinuse, cause_callmanager_failure, cause_callrejected, cause_callsplit, cause_chantypenimpl, cause_chanunacceptable, cause_ctimanager_failure, cause_destinationoutoforder, cause_destnummissandcnnotsub, cause_facilityrejected, cause_identifiedchandoesnotexist, cause_ienimpl, cause_inboundblindtransfer, cause_inboundconference, cause_inboundtransfer, cause_incomingcallbarred, cause_incompatibledestination, cause_interworkingunspecified, cause_invalidcallrefvalue, cause_invalidmessageunspecified, cause_invalidnumberformat, cause_invalidtransitnetset, cause_mandatoryiemergency, cause_mscncompatiblewcs, cause_mgtypencompatwcs, cause_mgtypenimpl, cause_netoutoforder, cause_nocircavail, cause_noerror, cause_noselecteduserclearing, cause_normalcallclearing, cause_normalunspecified, cause_normaluseridestination, cause_noroutetotransitnet, cause_nouserresponding, cause_numberchanged, cause_onlyrdivearercapavail, cause_outboundconference, cause_outboundtransfer, cause_protocolerrorunspecified, cause_qualofservnavail, cause_recoveryonexpiry, cause_redirected, cause_reqcallidhasbeencleared, cause_reqcircnavail, cause_reqfacilitynimpl, cause_reqfacilitynotsubscribed, cause_resourcenavail, cause_responsersetstatusquery, cause_servnotavailunspecified, cause_servoperationviolated, cause_servoroptnavailorimpl, cause_suspcallbutnotthisone, cause_switchingequipmentcongestion, cause_temporaryfailure, cause_unallocatednumber, cause_userbusy</td>
</tr>
</tbody>
</table>

### Methods inherited from interface CallEv

- `getCall()`

### Methods inherited from interface CiscoCallEv

- `getCiscoCause()`

### Methods inherited from interface Ev

- - -
Inherited Member Summary

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Fields

<table>
<thead>
<tr>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static final int ID</td>
</tr>
</tbody>
</table>

Methods

getConferenceControllerAddress()

public javax.telephony.Address getConferenceControllerAddress()

Returns the Address which currently acts as the conference controller for this call — the initiating call.

getConferencedCall()

public javax.telephony.Call getConferencedCall()

Returns the call that has been merged. This call is in the Call.INVALID state.

getFinalCall()

public javax.telephony.Call getFinalCall()

Returns the call that remains active after the conference is completed.

getHeldConferenceController()

public javax.telephony.TerminalConnection getHeldConferenceController()

Returns the TerminalConnection which currently acts as the conference controller for this call — the final call. This is the TerminalConnection that was in HELD state when conference was initiated. This method returns null if the conference controller is not being observed.
getTalkingConferenceController()

```java
public javax.telephony.TerminalConnection getTalkingConferenceController()
```

Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call. This is the TerminalConnection that was in TALKING state. This method returns null if the conference controller is not being observed.

## CiscoConferenceStartEv

### Declaration

```java
public interface CiscoConferenceStartEv extends CiscoCallEv
```

### All Superinterfaces

```java
javax.telephony.events.CallEv, CiscoCallEv, CiscoEv,
javax.telephony.events.Ev
```

### Description

The `CiscoConferenceStartEv` event indicates that a conference operation has started. This event is reported via the `CallControlCallObserver` interface.

### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td></td>
</tr>
</tbody>
</table>
CiscoConferenceStartEv

### Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.telephony.Address</td>
<td>getConferenceControllerAddress() Returns the Address which currently acts as the conference controller for this call — the initiating call.</td>
</tr>
<tr>
<td>javax.telephony.Call</td>
<td>getConferencedCall() Returns the call that will be conferenced.</td>
</tr>
<tr>
<td>javax.telephony.Call</td>
<td>getFinalCall() Returns the call that will remain active after the conference is completed.</td>
</tr>
<tr>
<td>javax.telephony.TerminalConnection</td>
<td>getHeldConferenceController() Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call.</td>
</tr>
<tr>
<td>javax.telephony.TerminalConnection</td>
<td>getTalkingConferenceController() Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call.</td>
</tr>
</tbody>
</table>

### Inherited Member Summary

Fields inherited from interface `CiscoCallEv`
### Inherited Member Summary

| CAUSE_ACCESSINFORMATIONDISCARDED, CAUSE_BCBPRESENTLYAVAIL, CAUSE_BCNAUTHORIZED, CAUSE_BEARERCAPNIMPL, CAUSE_CALLINGDELETED, CAUSE_CALLIDINUSE, CAUSE_CALLMANAGER_FAILURE, CAUSE_CALLREJECTED, CAUSE_CALLSPLIT, CAUSE_CHANACCEPTABLE, CAUSE_CALLTYPENIMPL, CAUSE_CALLMANAGER_FAILURE, CAUSE_DESTINATIONOUTOFORDER, CAUSE_DESTNUMMISSANDCNOTSUB, CAUSE_CALLIDREJECTED, CAUSE_IDENTIFIEDCHANDOESNOTEXIST, CAUSE_IENIMPL, CAUSE_INBOUNDBLINDTRANSFER, CAUSE_INBOUNDCALLCONFERENCE, CAUSE_INBOUNDCALLTRANSFER, CAUSE_INCOMINGCALLBARRED, CAUSE_INCOMPATABLEDESTINATION, CAUSE_INTERWORKINGUNSPECIFIED, CAUSE_INVALIDCALLREFVALUE, CAUSE_INVALIDIDNAMES, CAUSE_INVALIDMESSAGEUNSPECIFIED, CAUSE_INVALIDIDNUMBERFORMAT, CAUSE_INVALIDTRANSPORT, CAUSE_MANDATORYIEMISSING, CAUSE_MSGCOMPATABLEWCS, CAUSE_MSGTYPENIMPL, CAUSE_NETOUTOFORDER, CAUSE_NOANSWERFROMUSER, CAUSE_NOCALLSUSPENDED, CAUSE_NOINCOMPATIBILITY, CAUSE_NOERROR, CAUSE_NONZEROEDUSERCLEARING, CAUSE_NORMALCALLCLEARING, CAUSE_NORMALUNSPECIFIED, CAUSE_NOROUTEDESTINATION, CAUSE_NOROUTETOTRANSNET, CAUSE_NouserRESPONDING, CAUSE_NUMBERCHANGED, CAUSE_ONLYRDIVEARERCAPAVAIL, CAUSE_OUTBOUNDCONFERENCE, CAUSE_OUTBOUNDTRANSFER, CAUSE_PROTOCOLERRORUNSPECIFIED, CAUSE_QUALOFERNAVAIL, CAUSE_RECOVERYONTIMEREXPIRY, CAUSE_REDIRECTED, CAUSE_REQCALLIDHASBEENCLEARED, CAUSE_REQCIRCNAVAIL, CAUSE_REQFACILITYNIMPL, CAUSE_REQFACILITYNOTSUBSCRIBED, CAUSE_RESOURCESNAVAIL, CAUSE_RESPONSETOSTATUSENQUIRY, CAUSE_SERVNOTAVAILUNSPECIFIED, CAUSE_SERVOPTIONUNREL, CAUSE_SERVOPTIONNAVAILUNREL, CAUSE_SUSPCALLBUTNOTTHISONE, CAUSE_SWITCHINFACILITYCONGESTION, CAUSE_TEMPORARYFAILURE, CAUSE_UNALLOCATEDNUMBER, CAUSE_USERBUSY |

### Fields inherited from interface `Ev`

- `CAUSE_CALL_CANCELED`, `CAUSE_DEST NOT OBTAINABLE`, `CAUSE_INCOMPATIBLE_DESTINATION`, `CAUSE_LOCKOUT`, `CAUSE_NETWORK_CONGESTION`, `CAUSE_NETWORK NOT OBTAINABLE`, `CAUSE_NEW_CALL`, `CAUSE_NORMAL`, `CAUSE_RESOURCES NOT AVAILABLE`, `CAUSE_SNAPSHOT`, `CAUSE_UNKNOWN`, `META_CALL_ADDITIONAL_PARTY`, `META_CALL_ENDING`, `META_CALL_Merging`, `META_CALL_PROGRESS`, `META_CALL REMOVING_PARTY`, `META_CALL STARTING`, `META_CALL_TRANSFERRING`, `META_SNAPSHOT`, `META_UNKNOWN`
Fields

ID

public static final int ID

Methods

getConferenceControllerAddress()

public javax.telephony.Address getConferenceControllerAddress()

Returns the Address which currently acts as the conference controller for this call — the initiating call.

getConferencedCall()

public javax.telephony.Call getConferencedCall()

Returns the call that will be conferenced. This is the call that will be merged into the initiating call.

getFinalCall()

public javax.telephony.Call getFinalCall()

Returns the call that will remain active after the conference is completed. This is the call all calls are finally merged into.

getHeldConferenceController()

public javax.telephony.TerminalConnection getHeldConferenceController()

Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call. This is the TerminalConnection that was in HELD state. This method returns null if the conference controller is not being observed.
getTalkingConferenceController()

public javax.telephony.TerminalConnection
getTalkingConferenceController()

Returns the TerminalConnection which currently acts as the conference controller for this call — the initiating call. This is the TerminalConnection that was in TALKING state. This method returns null if the conference controller is not being observed.

CiscoConnection

Declaration

public interface CiscoConnection extends
javax.telephony.callcontrol.CallControlConnection,
CiscoObjectContainer

All Superinterfaces

javax.telephony.callcontrol.CallControlConnection,
CiscoObjectContainer, javax.telephony.Connection

Description

The CiscoConnection interface extends the CallControlConnection interface with additional CallManager-specific capabilities.

Applications can use the getReason method to obtain the reason for the creation of this Connection.

Member Summary

Fields
static int ADDRESS_SEARCH_SPACE
This indicates that the redirect should be done using the search space of the redirect controller’s address.

static int CALLED_ADDRESS_DEFAULT
This option indicates that the default behavior for Cisco JTAPI should apply.

static int CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION
This option indicates that the called Address should be reset to the redirect destination.

static int CALLED_ADDRESS_UNCHANGED
This option indicates that the called Address should remain unchanged after the redirect operation.

static int CALLINGADDRESS_SEARCH_SPACE
This indicates that the redirect should be done using the search space of the calling address.

static int DEFAULT_SEARCH_SPACE
This indicates that the redirect should be done using the search space that is the default for the implementation.

static int REASON_DIRECTCALL
This Connection was the result of a direct call.

static int REASON_FORWARDALL
This Connection was the result of unconditional forwarding.

static int REASON_FORWARDBUSY
This Connection was the result of a forwarding on busy.

static int REASON_FORWARDNOANSWER
This Connection was the result of a forwarding on no answer.

static int REASON_OUTBOUND
This Connection is an originating Connection, not a destination Connection.

static int REASON_REDIRECT
This Connection was the result of a redirection.

static int REASON_TRANSFERRED_CALL
This Connection was the result of a transfer.

static int REDIRECT_DROP_ON_FAILURE
This redirect mode instructs the implementation to perform redirect without checking the validity or availability of the destination.

static int REDIRECT_NORMAL
This redirect mode instructs the implementation to perform redirect if the destination is valid and available.

### Methods

```java
CiscoConnectionID getConnectionID()
```

CiscoConnectionID is a unique object that identifier among all ACTIVE calls with the same CallManagerID.
## Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getReason()</code></td>
<td>Returns the reason for the creation of this Connection.</td>
</tr>
<tr>
<td><code>java.lang.String park()</code></td>
<td>This method parks the call at a system park port and returns the address of the port.</td>
</tr>
</tbody>
</table>

```java
redirect(java.lang.String destinationAddress, int mode)
```

This method overloads the CallControlConnection.redirect() method.

```java
javax.telephony.Connection
    redirect(java.lang.String destinationAddress, int mode, int callingSearchSpace)
```

This method overloads the CallControlConnection.redirect() method.

```java
javax.telephony.Connection
callingSearchSpace, int calledAddressOption)
```

This method overloads the CallControlConnection.redirect() method.

## Inherited Member Summary

### Fields inherited from interface CallControlConnection

- ALERTING, DIALING, DISCONNECTED, ESTABLISHED, FAILED, IDLE, INITIATED, NETWORK_ALERTING, NETWORK_REACHED, OFFERED, OFFERING, QUEUED, UNKNOWN

### Fields inherited from interface Connection

- CONNECTED, INPROGRESS

### Methods inherited from interface CiscoObjectContainer

- `getObject()`, `setObject(Object)`

### Methods inherited from interface CallControlConnection

- `accept()`, `addToAddress(String)`, `getCallControlState()`, `park(String)`, `redirect(String)`, `reject()`

### Methods inherited from interface Connection

- `disconnect()`, `getAddress()`, `getCall()`, `getCapabilities()`, `getConnectionCapabilities(Terminal, Address)`, `getState()`, `getTerminalConnections()`
Fields

ADDRESS_SEARCH_SPACE
public static final int ADDRESS_SEARCH_SPACE
This indicates that the redirect should be done using the search space of the
redirect controller’s address.

CALLED_ADDRESS_DEFAULT
public static final int CALLED_ADDRESS_DEFAULT
This option indicates that the default behavior for Cisco JTAPI should apply.
Cisco JTAPI’s default behavior is the same as CALLED_ADDRESS_UNCHANGED.

CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION
public static final int CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION
This option indicates that the calledAddress should be reset to the redirect
destination.

CALLED_ADDRESS_UNCHANGED
public static final int CALLED_ADDRESS_UNCHANGED
This option indicates that the calledAddress should remain unchanged after
the redirect operation.

CALLINGADDRESS_SEARCH_SPACE
public static final int CALLINGADDRESS_SEARCH_SPACE
This indicates that the redirect should be done using the search space of the
calling address.

DEFAULT_SEARCH_SPACE
public static final int DEFAULT_SEARCH_SPACE
This indicates that the redirect should be done using the search space that is
the default for the implementation. The default is to use the calling address’s
search space.
REASON_DIRECTCALL
public static final int REASON_DIRECTCALL
This Connection was the result of a direct call.

REASON_FORWARDALL
public static final int REASON_FORWARDALL
This Connection was the result of unconditional forwarding.

REASON_FORWARDBUSY
public static final int REASON_FORWARDBUSY
This Connection was the result of a forwarding on busy.

REASON_FORWARDNOANSWER
public static final int REASON_FORWARDNOANSWER
This Connection was the result of a forwarding on no answer.

REASON_OUTBOUND
public static final int REASON_OUTBOUND
This Connection is an originating Connection, not a destination Connection.

REASON_REDIRECT
public static final int REASON_REDIRECT
This Connection was the result of a redirection.

REASON_TRANSFERREDCALL
public static final int REASON_TRANSFERREDCALL
This Connection was the result of a transfer.

REDIRECT_DROP_ON_FAILURE
public static final int REDIRECT_DROP_ON_FAILURE
This redirect mode instructs the implementation to perform redirect without checking the validity or availability of the destination. The original call will be dropped if the destination is not valid or if it’s busy.
REDIRECT_NORMAL

public static final int REDIRECT_NORMAL

This redirect mode instructs the implementation to perform redirect if the destination is valid and available. Otherwise, the request will return error. The original call will not be dropped on failure.

Methods

getConnectionID()

public com.cisco.jtapi.extensions.CiscoConnectionID getConnectionID()

CiscoConnectionID is a unique object that identifier among all ACTIVE calls with the same CallManagerID.

Returns: the CallID property of this Call

getReason()

public int getReason()

Returns the reason for the creation of this Connection.

In order to function properly, some applications need to know the reason why a Connection is created at an endpoint that the application is observing. For example, a voice mail application may want to know whether a caller is someone that wants to leave a message in a voice mailbox (REASON_FORWARDNOANSWER), or whether the caller is trying to access a voice mail box (REASON_DIRECTCALL).

The reason for a Connection’s creation may be any of the following constants:

- CiscoConnection.REASON_DIRECTCALL
- CiscoConnection.REASON_TRANSFERREDCALL
- CiscoConnection.REASON_FORWARDNOANSWER
- CiscoConnection.REASON_FORWARDBUSY
- CiscoConnection.REASON_FORWARDALL
- CiscoConnection.REASON_REDIRECT
- CiscoConnection.REASON_NORMAL

All of the reasons except for REASON_TRANSFERORIGINATION and REASONNORMAL are associated with inbound, or destination Connections. The REASON_NORMAL reason is associated with outbound, or originating Connections.

**Returns:** the reason for the creation of this Connection

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

This method parks the call at a system park port and returns the address of the port. The call can be unparked using this address.

**Errors:**
- javax.telephony.InvalidStateException
- javax.telephony.ResourceUnavailableException
- javax.telephony.PrivilegeViolationException
- javax.telephony.InvalidArgumentException

```java
public javax.telephony.Connection redirect(java.lang.String destinationAddress, int mode)
```

This method overloads the CallControlConnection.redirect() method. It takes a new parameter — redirectMode. When this parameter is:

1. CiscoConnection.REDIRECT_DROP_ON_FAILURE This mode instructs the implementation to perform redirect without checking the validity or availability of the destination. The original call will be dropped if the destination is not valid or if it's busy.

2. CiscoConnection.REDIRECT_NORMAL This mode instructs the implementation to perform redirect only after checking the validity or
availability of the destination. This is the same as the 
CallControlConnection.redirect() method. The original 
call will not be dropped on failure.

**Throws:**
javax.telephony.ResourceUnavailableException, 
javax.telephony.PrivilegeViolationException, 
javax.telephony.MethodNotSupportedException, 
javax.telephony.InvalidPartyException, 
javax.telephony.InvalidStateException

```java
redirect(String, int, int)
```

```java
public javax.telephony.Connection redirect(java.lang.String 
destinationAddress, int mode, 
int callingSearchSpace) 
throws InvalidStateException, InvalidPartyException, 
MethodNotSupportedException, PrivilegeViolationException, 
ResourceUnavailableException
```

This method overloads the CallControlConnection.redirect() method. It 
takes two new parameters — redirectMode and callingSearchSpace. The 
redirectMode is used to select which type of redirect to perform, and the 
callingSearchSpace is used to instruct the implementation to use either the 
calling party’s search space or the redirect controller’s search space.

The redirectMode parameter may be:
1. CiscoConnection.REDIRECT_DROP_ON_FAILURE 
2. CiscoConnection.REDIRECT_NORMAL

Read above for a description of what each of these means. 
The callingSearchSpace parameter may be:
1. CiscoConnection.DEFAULT_SEARCH_SPACE 
2. CiscoConnection.CALLINGADDRESS_SEARCH_SPACE 
3. CiscoConnection.ADDRESS_SEARCH_SPACE

Read above for a description of what each of these means. 
NOTE: The callingSearchSpace parameter may be specified when the 
redirect controller is a RouteAddress. It will be ignored for all other address types.
This method overloads the CallControlConnection.redirect() method. It takes three new parameters — redirectMode, callingSearchSpace and calledAddressOption. The redirectMode is used to select which type of redirect to perform, and the callingSearchSpace is used to instruct the implementation to use either the calling party’s search space or the redirect controller’s search space. The calledAddressOption parameter is used to decide whether to reset the original called fields or not.

The redirectMode parameter may be:

1. CiscoConnection.REDIRECT_DROP_ON_FAILURE
2. CiscoConnection.REDIRECT_NORMAL

Read above for a description of what each of these means.

The callingSearchSpace parameter may be:

1. CiscoConnection.DEFAULT_SEARCH_SPACE
2. CiscoConnection.CALLINGADDRESS_SEARCH_SPACE
3. CiscoConnection.ADDRESS_SEARCH_SPACE

Read above for a description of what each of these means.

NOTE: The callingSearchSpace parameter may be specified when the redirect controller is a RouteAddress. It will be ignored for all other address types.

* The calledAddressOption parameter may be:
1. CiscoConnection.CALLED_ADDRESS_DEFAULT
2. CiscoConnection.CALLED_ADDRESS_UNCHANGED
3. CiscoConnection.CALLED_ADDRESS_SET_TO_REDIRECT_DESTINATION

Read above for a description of what each of these means.

Throws:
javax.telephony.ResourceUnavailableException,
javax.telephony.PrivilegeViolationException,
javax.telephony.MethodNotSupportedException,
javax.telephony.InvalidPartyException,
javax.telephony.InvalidStateException

CiscoConnectionID

Declaration

public interface CiscoConnectionID extends CiscoObjectContainer

All Superinterfaces

CiscoObjectContainer

Description

The CiscoConnectionID object represents a unique object associated with each connection. Applications may use the object itself or the integer representation of the object returned by the intValue() method.
Member Summary

Methods

CiscoConnection getConnection()

int intValue()

Returns an integer representation of this object, currently the CallManager CallLeg ID.

Inherited Member Summary

Methods inherited from interface CiscoObjectContainer

getObject(), setObject(Object)

Methods

getConnection()

public com.cisco.jtapi.extensions.CiscoConnection

getConnection()

intValue()

public int intValue()

Returns an integer representation of this object, currently the CallManager CallLeg ID.

Returns: an integer representation of this object

CiscoConsultCall

Declaration

public interface CiscoConsultCall extends CiscoCall
### All Superinterfaces

javax.telephony.Call,  
javax.telephony.callcontrol.CallControlCall,  
CiscoCall, CiscoObjectContainer

### Description

The `CiscoConsultCall` interface extends the `CiscoCall` interface to expose certain properties of Calls that have been created as part of a consultative transfer or consultative conference.

**See Also:**  
javax.telephony.Call

### Member Summary

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>consultWithoutMedia(javax.telephony.TerminalConnection tc, java.lang.String dialedDigits)</code></td>
<td>From CallEvent perspective, this method behaves similar to <code>CallControlCall.consult(TerminalConnection tc, String dialedDigits)</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getConsultingTerminalConnection()</code></td>
<td>Returns the consulting TerminalConnection that was used to create this <code>CiscoConsultCall</code>.</td>
</tr>
</tbody>
</table>

### Inherited Member Summary

**Fields inherited from interface Call**

ACTIVE, IDLE, INVALID
Inherited Member Summary

Methods inherited from interface Call

addObserver(CallObserver), connect(Terminal, Address, String),
getCallCapabilities(Terminal, Address),
getConnections(), getObservers(), getProvider(), getState(),
removeObserver(CallObserver)

Methods inherited from interface CallControlCall

addParty(String), conference(Call), consult(TerminalConnection),
consult(TerminalConnection), drop(), getCalledAddress(), getCallingAddress(),
getCallingTerminal(), getConferenceController(), getConferenceEnable(),
getLastRedirectedAddress(), getTransferController(), getTransferEnable(),
offHook(Address, Terminal), setConferenceController(TerminalConnection),
setConferenceEnable(boolean), setTransferController(TerminalConnection),
setTransferEnable(boolean), transfer(String), transfer(String)

Methods inherited from interface CiscoCall

getCallID(), getCurrentCalledAddress(), getCurrentCalledPartyDisplayName(),
getCurrentCallingAddress(), getCurrentCallingPartyDisplayName()

Methods inherited from interface CiscoObjectContainer

getObject(), setObject(Object)

Methods

consultWithoutMedia(TerminalConnection, String)

public javax.telephony.Connection[
    consultWithoutMedia(javax.telephony.TerminalConnection tc, java.lang.String dialedDigits)
    throws InvalidStateException, InvalidArgumentException, MethodNotSupportedException, ResourceUnavailableException, PrivilegeViolationException, InvalidPartyException

From CallEvent perspective, this method behaves similar to
CallControlCall.consult(TerminalConnection tc, String dialedDigits). Creates a consultation between this Call and
an active Call without establishing the media. This consult call may only be transferred, not conferenced.

**The Consultation Purpose**
This method does not support if invoked with `CallControlCall.setConferneceEnable()`. It only supports if invoked with `CallControlCall.setTransferEnable()`.

**Throws:**
- `javax.telephony.InvalidPartyException`
- `javax.telephony.PrivilegeViolationException`
- `javax.telephony.ResourceUnavailableException`
- `javax.telephony.MethodNotSupportedException`
- `javax.telephony.InvalidStateException`
- `javax.telephony.InvalidArgumentException`

`getConsultingTerminalConnection()`

```java
public javax.telephony.TerminalConnection
getConsultingTerminalConnection()
```

Returns the consulting TerminalConnection that was used to create this CiscoConsultCall.

If this Call was created as part of a consultative transfer or consultative conference, the TerminalConnection which was used to perform the consultation on the original Call is returned by the `getConsultingTerminalConnection` method. This may be useful to applications that wish to correlate a ConsultCall with its original Call. Note that the original Call does not have any methods which may be used to determine the ConsultCall, if any, to which it is related.

**Returns:** null if this Call is not the result of a consultation, or the consulting TerminalConnection of the original Call if this Call is the result of a consultation.
CiscoConsultCallActiveEv

Declaration

```java
public interface CiscoConsultCallActiveEv extends CiscoCallEv,
                javax.telephony.events.CallActiveEv,
```

All Superinterfaces

- javax.telephony.events.CallActiveEv
- javax.telephony.events.CallEv
- CiscoCallEv
- CiscoEv
- javax.telephony.events.Ev

Description

The `CiscoConsultCallActiveEv` event interface extends the JTAPI `CallActiveEv`. It indicates that the state of the Call object has changed to `Call.ACTIVE` and that the call was initiated as a result of a consultative transfer or consultative conference operation (manual or programmatic). Applications can obtain the consulting TerminalConnection on the original (consulting) call by using the `CiscoConsultCall.getConsultingTerminalConnection` method.

This event is reported to applications via the `CallObserver` interface.

See Also:

- javax.telephony.Call, javax.telephony.CallObserver,
- javax.telephony.events.CallActiveEv

Member Summary

Fields
### Member Summary

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>static int</td>
</tr>
<tr>
<td>getHeldTerminalConnection()</td>
<td>Returns the consulting TerminalConnection that was used to create this CiscoConsultCall.</td>
</tr>
</tbody>
</table>

### Inherited Member Summary

Fields inherited from interface `CiscoCallEv`:
- `CAUSE_ACCESSINFORMATIONDISCARDED`, `CAUSE_BCBPRESENTLYAVAIL`, `CAUSE_BCNAUTHORIZED`, `CAUSE_BEARERCAPNIMPL`, `CAUSE_CALLBEINGDELIVERED`, `CAUSE_CALLIDUNUSED`, `CAUSE_CALLMANAGER_FAILURE`, `CAUSE_CALLREJECTED`, `CAUSE_CALLSPLIT`, `CAUSE_CHANYPENIMPL`, `CAUSE_CHANUNACCEPTABLE`, `CAUSE_CTIMANAGER_FAILURE`, `CAUSE_DESTINATIONOUTOFORDER`, `CAUSE_DESTNUMMISSANDDCNOTSUB`, `CAUSE_FACILITYREJECTED`, `CAUSE_IDENTIFIEDCHANDOESNOTEXIST`, `CAUSE_IENIMPL`, `CAUSE_INBOUNDBLINDTRANSFER`, `CAUSE_INBOUNDCONFERENCE`, `CAUSE_INBOUNDTRANSFER`, `CAUSE_INCOMINGCALLBARRED`, `CAUSE_INCOMPATABLEDESTINATION`, `CAUSE_INTERWORKINGUNSPECIFIED`, `CAUSE_INVALIDCALLREFVALUE`, `CAUSE_INVALIDMESSAGEUNSPECIFIED`, `CAUSE_INVALIDNUMBERFORMAT`, `CAUSE_INVALIDTRANSITNETSEL`, `CAUSE_MANDATORYIEMISSING`, `CAUSE_MSGNCOMPATABLEWCS`, `CAUSE_MSGTYPENCOMPATWCS`, `CAUSE_MSGTYPENIMPL`, `CAUSE_NETOUTOFORDER`, `CAUSE_NOANSWERFROMUSER`, `CAUSE_NOCALLSUSPENDED`, `CAUSE_NOCIRCNAVAIL`, `CAUSE_NONERROR`, `CAUSE_NONSELECTEDUSERCLEARING`, `CAUSE_NORMALCALLCLEARING`, `CAUSE_NORMALUNSPECIFIED`, `CAUSE_NOROUTETODESTINATION`, `CAUSE_NOROUTETOTRANSITNET`, `CAUSE_NOUSERRESPONDING`, `CAUSE_NUMBERCHANGED`, `CAUSE_ONLYDIVEARERCAPAVAIL`, `CAUSE_OUTBOUNDCONFERENCE`, `CAUSE_OUTBOUNDTRANSFER`, `CAUSE_PROTOCOLERRORUNSPECIFIED`, `CAUSE_QUALOFSERVNAVAIL`, `CAUSE_RECOVERYONTIMEREXPIRY`, `CAUSE_REDIREREDIRECTED`, `CAUSE_REQCALLIDHASBECOMENAVAIL`, `CAUSE_REQFACILITYNOTSUBSCRIBED`, `CAUSE_RESOURCESSNAVAIL`, `CAUSE_RESPONSETOSTATUSENQUIRY`, `CAUSE_SERVNOTAVAILUNSPECIFIED`, `CAUSE_SERVOPERATIONVIOLATED`, `CAUSE_SERVOROPTNAVAILUNSPECIFIED`, `CAUSE_SUSPCALLBUTNOTTHISONE`, `CAUSE_SWITCHINGEQUIPMENTCONGESTION`, `CAUSE_TEMPORARYFAILURE`, `CAUSE_UNALLOCATEDNUMBER`, `CAUSE_USERBUSY`
Inherited Member Summary

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface CallEv

getCall()

Methods inherited from interface CiscoCallEv

getCiscoCause()

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Fields

ID

public static final int ID

Methods

getHeldTerminalConnection()

public javax.telephony.TerminalConnection

getheldTerminalConnection()

Deprecated. replaced by
CiscoConsultCall.getConsultingTerminalConnection ()

Returns the consulting TerminalConnection that was used to create this CiscoConsultCall.

This may be useful to applications that wish to correlate a consultation Call with its original Call. Note that the original Call does not have any methods
which may be used to determine the consultation Call, if any, to which it is related.

**Returns:** the consulting TerminalConnection of the Call that created the Call referenced by this event

### CiscoEv

**Declaration**

```java
public interface CiscoEv extends javax.telephony.events.Ev
```

### All Superinterfaces

```java
javax.telephony.events.Ev
```

### All Known Subinterfaces

- `CiscoAddrCreatedEv`, `CiscoAddrEv`, `CiscoAddrInServiceEv`, `CiscoAddrOutOfServiceEv`, `CiscoAddrRemovedEv`, `CiscoCallEv`, `CiscoConferenceEndEv`, `CiscoConferenceStartEv`, `CiscoConsultCallActiveEv`, `CiscoOutOfServiceEv`, `CiscoProvCallParkEv`, `CiscoProvEv`, `CiscoProvFeatureEv`, `CiscoProvFeatureUnRegisteredEv`, `CiscoRTPInputStartedEv`, `CiscoRTPInputStoppedEv`, `CiscoRTPOutputStartedEv`, `CiscoRTPOutputStoppedEv`, `CiscoTermCreatedEv`, `CiscoTermDataEv`, `CiscoTermEv`, `CiscoTermInServiceEv`, `CiscoTermOutOfServiceEv`, `CiscoTermRemovedEv`, `CiscoTransferEndEv`, `CiscoTransferStartEv`
Description

The CiscoEv interface, which extends JTAPI’s core javax.telephony.events.Ev interface, serves as the base interface for all Cisco-extended JTAPI events. Every event in this package extends this interface, directly or indirectly.

See Also:
javax.telephony.events.Ev

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()
CiscoG711MediaCapability

Description

The CiscoG711MediaCapability object specifies the properties for a G.711 encoded RTP stream. Applications that support G.711 media termination use this object to specify their preferred packet size when registering a CiscoMediaTerminal.

The default packet size is thirty milliseconds.

Member Summary

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int FRAMESIZE_SIXTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 60 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_THIRTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 30 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_TWENTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 20 millisecond packets</td>
</tr>
</tbody>
</table>

Constructors

- CiscoG711MediaCapability()
  Constructs a CiscoG711MediaCapability object with a default thirty millisecond packet size.
- CiscoG711MediaCapability(int maxFramesPerPacket)
  Constructs a CiscoG711MediaCapability object with the specified packet size.

Inherited Member Summary

Fields inherited from class CiscoMediaCapability

- G711_64K_30_MILLISECONDS, G723_6K_30_MILLISECONDS, G729_30_MILLISECONDS, GSM_80_MILLISECONDS

Methods inherited from class CiscoMediaCapability
### Inherited Member Summary

<table>
<thead>
<tr>
<th>Methods inherited from class Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(), wait()</td>
</tr>
</tbody>
</table>

### Fields

**FRAMESIZE_SIXTY_MILLISECOND_PACKET**  
public static final int `FRAMESIZE_SIXTY_MILLISECOND_PACKET`  
The frames-per-packet value for 60 millisecond packets

**FRAMESIZE_THIRTY_MILLISECOND_PACKET**  
public static final int `FRAMESIZE_THIRTY_MILLISECOND_PACKET`  
The frames-per-packet value for 30 millisecond packets

**FRAMESIZE_TWENTY_MILLISECOND_PACKET**  
public static final int `FRAMESIZE_TWENTY_MILLISECOND_PACKET`  
The frames-per-packet value for 20 millisecond packets

### Constructors

**CiscoG711MediaCapability()**  
public `CiscoG711MediaCapability()`  
Constructs a `CiscoG711MediaCapability` object with a default thirty millisecond packet size.

**CiscoG711MediaCapability(int)**  
public `CiscoG711MediaCapability(int maxFramesPerPacket)`
CiscoG723MediaCapability

Declaration

```java
public class CiscoG723MediaCapability extends CiscoMediaCapability
java.lang.Object
    +-- com.cisco.jtapi.extensions.CiscoMediaCapability
        +-- com.cisco.jtapi.extensions.CiscoG723MediaCapability
```

Description

The `CiscoG723MediaCapability` object specifies the properties for a G.723 encoded RTP stream. Applications that support G.723 media termination use this object to specify their preferred packet size and bit rate when registering a `CiscoMediaTerminal`.

The default packet size is thirty milliseconds and the default bit rate is 6.4k.

### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int FRAMESIZE_SIXTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 60 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_THIRTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 30 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_TWENTY_MILLISECOND_PACKET</td>
<td>The frames-per-packet value for 20 millisecond packets</td>
</tr>
</tbody>
</table>

Constructors
### Member Summary

**Fields**

```java
public static final int FRAMESIZE_SIXTY_MILLISECOND_PACKET
```

The frames-per-packet value for 60 millisecond packets.

**Methods**

```java
public static final int FRAMESIZE_SIXTY_MILLISECOND_PACKET
```

The frames-per-packet value for 60 millisecond packets.

```java
public class CiscoG723MediaCapability
```

Constructs a `CiscoG723MediaCapability` object with a default thirty millisecond packet size and 6.4k bit rate.

```java
public class CiscoG723MediaCapability(int maxFramesPerPacket, int bitRate)
```

Constructs a `CiscoG723MediaCapability` object with the specified packet size and bit rate.

```java
int getBitRate()
```

Returns the bit rate specified by this capability object.

```java
java.lang.String toString()
```

### Inherited Member Summary

**Fields inherited from class `CiscoMediaCapability`**

- `G711_64K_30_MILLISECONDS`, `G723_6K_30_MILLISECONDS`, `G729_30_MILLISECONDS`, `GSM_80_MILLISECONDS`

**Methods inherited from class `CiscoMediaCapability`**

- `getMaxFramesPerPacket()`, `getPayloadType()`

**Methods inherited from class `Object`**

- `clone()`, `equals(Object)`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `wait()`, `wait()`, `wait()`
FRAMESIZE_THIRTY_MILLISECOND_PACKET

public static final int FRAMESIZE_THIRTY_MILLISECOND_PACKET

The frames-per-packet value for 30 millisecond packets

FRAMESIZE_TWENTY_MILLISECOND_PACKET

public static final int FRAMESIZE_TWENTY_MILLISECOND_PACKET

The frames-per-packet value for 20 millisecond packets

Constructors

CiscoG723MediaCapability()

public CiscoG723MediaCapability()

Constructs a CiscoG723MediaCapability object with a default thirty millisecond packet size and 6.4k bit rate.

CiscoG723MediaCapability(int, int)

public CiscoG723MediaCapability(int maxFramesPerPacket,
int bitRate)

Constructs a CiscoG723MediaCapability object with the specified packet size and bit rate.

Methods

getBitRate()

public int getBitRate()

Returns the bit rate specified by this capability object.

Returns: a bit rate from the RTPBitRate interface

toString()

public java.lang.String toString()

Overrides: toString in class CiscoMediaCapability
CiscoG729MediaCapability

Declaration

public class CiscoG729MediaCapability extends CiscoMediaCapability

java.lang.Object
    |-- com.cisco.jtapi.extensions.CiscoMediaCapability
        |-- com.cisco.jtapi.extensions.CiscoG729MediaCapability

Description

The CiscoG729MediaCapability object specifies the properties for a G.729 encoded RTP stream. Applications that support G.729 media termination use this object to specify their preferred packet size when registering a CiscoMediaTerminal.

The default packet size is thirty milliseconds.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int FRAMESIZE_SIXTY_MILLISECOND_PACKET</td>
</tr>
<tr>
<td>The frames-per-packet value for 60 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_THIRTY_MILLISECOND_PACKET</td>
</tr>
<tr>
<td>The frames-per-packet value for 30 millisecond packets</td>
</tr>
<tr>
<td>static int FRAMESIZE_TWENTY_MILLISECOND_PACKET</td>
</tr>
<tr>
<td>The frames-per-packet value for 20 millisecond packets</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoG729MediaCapability()</td>
</tr>
<tr>
<td>Constructs a CiscoG729MediaCapability object with a default G729 payload and thirty millisecond packet size.</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

Member Summary

CiscoG729MediaCapability(int payload, int maxFramesPerPacket)
Constructs a CiscoG729MediaCapability object with the specified packet size and payload.

Inherited Member Summary

Fields inherited from class CiscoMediaCapability
G711_64K_30_MILLISECONDS, G723_6K_30_MILLISECONDS, G729_30_MILLISECONDS, GSM_80_MILLISECONDS

Methods inherited from class CiscoMediaCapability
getMaxFramesPerPacket(), getPayloadType(), toString()

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(), wait()

Fields

FRAMESIZE_SIXTY_MILLISECOND_PACKET
public static final int FRAMESIZE_SIXTY_MILLISECOND_PACKET
The frames-per-packet value for 60 millisecond packets

FRAMESIZE_THIRTY_MILLISECOND_PACKET
public static final int FRAMESIZE_THIRTY_MILLISECOND_PACKET
The frames-per-packet value for 30 millisecond packets

FRAMESIZE_TWENTY_MILLISECOND_PACKET
public static final int FRAMESIZE_TWENTY_MILLISECOND_PACKET
The frames-per-packet value for 20 millisecond packets
Cisco JTAPI Developer Guide for Cisco CallManager 3.3(2)

Chapter 2  Cisco JTAPI Implementation

CiscoGSMMediaCapability

Constructors

CiscoG729MediaCapability()

public CiscoG729MediaCapability()

Constructs a CiscoG729MediaCapability object with a default G729 payload and thirty millisecond packet size.

CiscoG729MediaCapability(int, int)

public CiscoG729MediaCapability(int payload,
int maxFramesPerPacket)

Constructs a CiscoG729MediaCapability object with the specified packet size and payload. Choice of payload is specified in CiscoRTPPayload with the options CiscoRTPPayload.G729 and CiscoRTPPayload.G729ANNEXA

CiscoGSMMediaCapability

Declaration

public class CiscoGSMMediaCapability extends CiscoMediaCapability

java.lang.Object
    +--com.cisco.jtapi.extensions.CiscoMediaCapability
        +--com.cisco.jtapi.extensions.CiscoGSMMediaCapability

Description

The CiscoGSMMediaCapability object specifies the properties for a GSM encoded RTP stream. Applications that support GSM media termination use this
object to specify their preferred packet size when registering a CiscoMediaTerminal.
The default packet size is thirty milliseconds.

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td>static int FRAMESIZE_EIGHTY_MILLISECOND_PACKET</td>
</tr>
<tr>
<td>The frames-per-packet value for 30 millisecond packets</td>
</tr>
<tr>
<td><strong>Constructors</strong></td>
</tr>
<tr>
<td>CiscoGSMMediaCapability()</td>
</tr>
<tr>
<td>Constructs a CiscoGSMMediaCapability object with a default eighty millisecond packet size.</td>
</tr>
<tr>
<td>CiscoGSMMediaCapability(int maxFramesPerPacket)</td>
</tr>
<tr>
<td>Constructs a CiscoGSMMediaCapability object with the specified packet size.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields inherited from class</strong> CiscoMediaCapability</td>
</tr>
<tr>
<td>G711_64K_30_MILLISECONDS, G723_6K_30_MILLISECONDS, G729_30_MILLISECONDS, GSM_80_MILLISECONDS</td>
</tr>
<tr>
<td><strong>Methods inherited from class</strong> CiscoMediaCapability</td>
</tr>
<tr>
<td>getMaxFramesPerPacket(), getPayloadType(), toString()</td>
</tr>
<tr>
<td><strong>Methods inherited from class</strong> Object</td>
</tr>
<tr>
<td>clone(), equals(Object), finalize(), getClass(), notify(), notifyAll(), wait(), wait(), wait()</td>
</tr>
</tbody>
</table>
Fields

```java
public static final int FRAMESIZE_EIGHTY_MILLISECOND_PACKET
The frames-per-packet value for 30 millisecond packets
```

Constructors

```java
public CiscoGSMMediaCapability()
Constructs a CiscoGSMMediaCapability object with a default eighty millisecond packet size.
```

```java
public CiscoGSMMediaCapability(int maxFramesPerPacket)
Constructs a CiscoGSMMediaCapability object with the specified packet size.
```

CiscoJtapiException

Declaration

```java
public interface CiscoJtapiException
```

Description

The CiscoJtapiException interface defines CTI error codes. These are the error codes that may be returned by CTI requests. All the JTAPI exceptions have been extended to implement this interface. The Error codes can be got by casting the exception to CiscoJtapiException and calling the method getErrorCode(). if e is any exception caught in an application, then try to see if
its an instance of CiscoJtapiException.

```java
try {
    // some code here }
    catch (Exception e) {
        if (e instanceof CiscoJtapiException)
            CiscoJtapiException ce = com.cisco.cti.client.CTIFAILURE.(CiscoJtapiException) e
    }
```

### Member Summary

#### Fields

- static int ASSOCIATED_LINE_NOT_OPEN
- static int CALL_ALREADY_EXISTS
- static int CALLHANDLE_NOTINCOMINGCALL
- static int CALLHANDLE_UNKNOWN_TO_LINECONTROL
- static int CANNOT_OPEN_DEVICE
- static int CANNOT_TERMINATE_MEDIA_ON_PHONE
- static int CFWDALL_ALREADY_OFF
- static int CFWDALL_ALREADY_SET
- static int CFWDALL_DESTN_INVALID
- static int CLUSTER_LINK_FAILURE
- static int COMMAND_NOT_IMPLEMENTED_ON_DEVICE
- static int CONFERENCE_ALREADY_PRESENT
- static int CONFERENCE_FAILED
- static int CONFERENCE_FULL
- static int CONFERENCE_INACTIVE
- static int CONFERENCE_INVALID_PARTICIPANT
- static int CTIERR_INCORRECT_MEDIA_CAPABILITY
- static int CTIERR_INVALID_DTMFDIGITS
- static int CTIERR_OPERATION_FAILED_QUIETCLEAR
- static int DARES_INVALID_REQ_TYPE
### Member Summary

<table>
<thead>
<tr>
<th>Member Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int DATA_SIZE_LIMIT_EXCEEDED</td>
<td></td>
</tr>
<tr>
<td>static int DB_ERROR</td>
<td></td>
</tr>
<tr>
<td>static int DB_ILLEGAL_DEVICE_TYPE</td>
<td></td>
</tr>
<tr>
<td>static int DB_NO_MORE_DEVICES</td>
<td></td>
</tr>
<tr>
<td>static int DESTINATION_BUSY</td>
<td></td>
</tr>
<tr>
<td>static int DESTINATION_UNKNOWN</td>
<td></td>
</tr>
<tr>
<td>static int DEVICE_ALREADY_REGISTERED</td>
<td></td>
</tr>
<tr>
<td>static int DEVICE_NOT_OPEN</td>
<td></td>
</tr>
<tr>
<td>static int DEVICE_OUT_OF_SERVICE</td>
<td></td>
</tr>
<tr>
<td>static int DIGIT_GENERATION_ALREADY_IN_PROGRESS</td>
<td></td>
</tr>
<tr>
<td>static int DIGIT_GENERATION_CALLSTATE_CHANGED</td>
<td></td>
</tr>
<tr>
<td>static int DIGIT_GENERATION_WRONG_CALL_HANDLE</td>
<td></td>
</tr>
<tr>
<td>static int DIGIT_GENERATION_WRONG_CALL_STATE</td>
<td></td>
</tr>
<tr>
<td>static int DIRECTORY_LOGIN_FAILED</td>
<td></td>
</tr>
<tr>
<td>static int DIRECTORY_LOGIN_NOT_ALLOWED</td>
<td></td>
</tr>
<tr>
<td>static int DIRECTORY_TEMPORARY_UNAVAILABLE</td>
<td></td>
</tr>
<tr>
<td>static int EXISTING_FIRSTPARTY</td>
<td></td>
</tr>
<tr>
<td>static int HOLDFAILED</td>
<td></td>
</tr>
<tr>
<td>static int ILLEGAL_CALLINGPARTY</td>
<td></td>
</tr>
<tr>
<td>static int ILLEGAL_CALLSTATE</td>
<td></td>
</tr>
<tr>
<td>static int ILLEGAL_HANDLE</td>
<td></td>
</tr>
<tr>
<td>static int ILLEGAL_MESSAGE_FORMAT</td>
<td></td>
</tr>
<tr>
<td>static int INCOMPATIBLE_PROTOCOL_VERSION</td>
<td></td>
</tr>
<tr>
<td>static int INVALID_LINE_HANDLE</td>
<td></td>
</tr>
<tr>
<td>static int INVALID_RING_OPTION</td>
<td></td>
</tr>
<tr>
<td>static int LINE_INFO_DOES_NOT_EXIST</td>
<td></td>
</tr>
<tr>
<td>static int LINE_NOT_PRIMARY</td>
<td></td>
</tr>
<tr>
<td>static int LINECONTROL_FAILURE</td>
<td></td>
</tr>
<tr>
<td>static int MAX_NUMBER_OF_CTI_CONNECTIONS_REACHED</td>
<td></td>
</tr>
<tr>
<td>static int MSGWAITING_DESTN_INVALID</td>
<td></td>
</tr>
<tr>
<td>static int NO_ACTIVE_DEVICE_FOR_THIRDPARTY</td>
<td></td>
</tr>
<tr>
<td>static int NO_CONFERENCE_BRIDGE</td>
<td></td>
</tr>
<tr>
<td>static int NOT_INITIALIZED</td>
<td></td>
</tr>
<tr>
<td>static int PROTOCOL_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>static int PROVIDER_ALREADY_OPEN</td>
<td></td>
</tr>
<tr>
<td>static int PROVIDER_CLOSED</td>
<td></td>
</tr>
<tr>
<td>static int PROVIDER_NOT_OPEN</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_CALL_TABLE_FULL</td>
<td></td>
</tr>
</tbody>
</table>
### Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int REDIRECT_CALL_DESTINATION_BUSY</td>
<td>The CTI error codes.</td>
</tr>
<tr>
<td>static int REDIRECT_CALL_DESTINATION_OUT_OF_ORDER</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_DIGIT_ANALYSIS_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_DOES_NOT_EXIST</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_INCOMPATIBLE_STATE</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_MEDIA_CONNECTION_FAILED</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_NORMAL_CLEARING</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_ORIGINATOR_ABANDONED</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_PARTY_TABLE_FULL</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_PENDING_REDIRECT_TRANSACTION</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_PROTOCOL_ERROR</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_UNKNOWN_DESTINATION</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_UNKNOWN_ERROR</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_UNKNOWN_PARTY</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALL_UNRECOGNIZED_MANAGER</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_CALLINFO_ERR</td>
<td></td>
</tr>
<tr>
<td>static int REDIRECT_ERR</td>
<td></td>
</tr>
<tr>
<td>static int RETRIEVEFAILED</td>
<td></td>
</tr>
<tr>
<td>static int RETRIEVEFAILED_ACTIVE_CALL_ON_LINE</td>
<td></td>
</tr>
<tr>
<td>static int SSAPI_NOT_REGISTERED</td>
<td></td>
</tr>
<tr>
<td>static int TEMPORARY_FAILURE</td>
<td></td>
</tr>
<tr>
<td>static int TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFER_INACTIVE</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFERFAILED</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFERFAILED_CALLCONTROL_TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFERFAILED_DESTINATION_BUSY</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFERFAILED_DESTINATION_UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>static int TRANSFERFAILED_OUTSTANDING_TRANSFER</td>
<td></td>
</tr>
<tr>
<td>static int UNDEFINED_LINE</td>
<td></td>
</tr>
<tr>
<td>static int UNKNOWN_GLOBAL_CALL_HANDLE</td>
<td></td>
</tr>
<tr>
<td>static int UNRECOGNIZABLE_PDU</td>
<td></td>
</tr>
<tr>
<td>static int UNSPECIFIED</td>
<td></td>
</tr>
</tbody>
</table>

### Methods

- `int getErrorCode()`: Returns the errorCode for this exception
- `java.lang.String getErrorDescription()`: This method returns the detail description of the errorCode
Chapter 2      Cisco JTAPI Implementation

CiscoJtapiException

### Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String getErrorDescription(int errorCode)</td>
<td>This method returns the detail description of the errorCode</td>
</tr>
<tr>
<td>java.lang.String getErrorName()</td>
<td>This method returns an exception in the string format.</td>
</tr>
<tr>
<td>java.lang.String getErrorName(int errorCode)</td>
<td>This method will return an exception in the string format.</td>
</tr>
</tbody>
</table>

### Fields

- **ASSOCIATED_LINE_NOT_OPEN**
  - public static final int ASSOCIATED_LINE_NOT_OPEN

- **CALL_ALREADY_EXISTS**
  - public static final int CALL_ALREADY_EXISTS

- **CALLHANDLE_NOTINCOMINGCALL**
  - public static final int CALLHANDLE_NOTINCOMINGCALL

- **CALLHANDLE_UNKNOWN_TO_LINECONTROL**
  - public static final int CALLHANDLE_UNKNOWN_TO_LINECONTROL

- **CANNOT_OPEN_DEVICE**
  - public static final int CANNOT_OPEN_DEVICE

- **CANNOT_TERMINATE_MEDIA_ON_PHONE**
  - public static final int CANNOT_TERMINATE_MEDIA_ON_PHONE

- **CFWDALL_ALREADY_OFF**
  - public static final int CFWDALL_ALREADY_OFF

- **CFWDALL_ALREADY_SET**
  - public static final int CFWDALL_ALREADY_SET

- **CFWDALL DESTN INVALID**
  - public static final int CFWDALL DESTN INVALID
public static final int CLUSTER_LINK_FAILURE

public static final int COMMAND_NOT_IMPLEMENTED_ON_DEVICE

public static final int CONFERENCE_ALREADY_PRESENT

public static final int CONFERENCE_FAILED

public static final int CONFERENCE_FULL

public static final int CONFERENCE_INACTIVE

public static final int CONFERENCE_INVALID_PARTICIPANT

public static final int CTIERR_INCORRECT_MEDIA_CAPABILITY

public static final int CTIERR_INVALID_DTMFDIGITS

public static final int CTIERR_OPERATION_FAILED_QUIETCLEAR

public static final int DARES_INVALID_REQ_TYPE

public static final int DATA_SIZE_LIMIT_EXCEEDED
<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_ERROR</td>
<td>public static final int DB_ERROR</td>
</tr>
<tr>
<td>DB_ILLEGAL_DEVICE_TYPE</td>
<td>public static final int DB_ILLEGAL_DEVICE_TYPE</td>
</tr>
<tr>
<td>DB_NO_MORE_DEVICES</td>
<td>public static final int DB_NO_MORE_DEVICES</td>
</tr>
<tr>
<td>DESTINATION_BUSY</td>
<td>public static final int DESTINATION_BUSY</td>
</tr>
<tr>
<td>DESTINATION_UNKNOWN</td>
<td>public static final int DESTINATION_UNKNOWN</td>
</tr>
<tr>
<td>DEVICE_ALREADY_REGISTERED</td>
<td>public static final int DEVICE_ALREADY_REGISTERED</td>
</tr>
<tr>
<td>DEVICE_NOT_OPEN</td>
<td>public static final int DEVICE_NOT_OPEN</td>
</tr>
<tr>
<td>DIGIT_GENERATION_ALREADY_IN_PROGRESS</td>
<td>public static final int DIGIT_GENERATION_ALREADY_IN_PROGRESS</td>
</tr>
<tr>
<td>DIGIT_GENERATION_CALLSTATE_CHANGED</td>
<td>public static final int DIGIT_GENERATION_CALLSTATE_CHANGED</td>
</tr>
<tr>
<td>DIGIT_GENERATION_WRONG_CALL_HANDLE</td>
<td>public static final int DIGIT_GENERATION_WRONG_CALL_HANDLE</td>
</tr>
<tr>
<td>DIGIT_GENERATION_WRONG_CALL_STATE</td>
<td>public static final int DIGIT_GENERATION_WRONG_CALL_STATE</td>
</tr>
</tbody>
</table>
DIRECTORY_LOGIN_FAILED
  public static final int DIRECTORY_LOGIN_FAILED

DIRECTORY_LOGIN_NOT_ALLOWED
  public static final int DIRECTORY_LOGIN_NOT_ALLOWED

DIRECTORY_TEMPORARY_UNAVAILABLE
  public static final int DIRECTORY_TEMPORARY_UNAVAILABLE

EXISTING_FIRSTPARTY
  public static final int EXISTING_FIRSTPARTY

HOLDFAILED
  public static final int HOLDFAILED

ILLEGAL_CALLINGPARTY
  public static final int ILLEGAL_CALLINGPARTY

ILLEGAL_CALLSTATE
  public static final int ILLEGAL_CALLSTATE

ILLEGAL_HANDLE
  public static final int ILLEGAL_HANDLE

ILLEGAL_MESSAGE_FORMAT
  public static final int ILLEGAL_MESSAGE_FORMAT

INCOMPATIBLE_PROTOCOL_VERSION
  public static final int INCOMPATIBLE_PROTOCOL_VERSION

INVALID_LINE_HANDLE
  public static final int INVALID_LINE_HANDLE

INVALID_RING_OPTION
  public static final int INVALID_RING_OPTION
LINE_INFO_DOES_NOT_EXIST
    public static final int LINE_INFO_DOES_NOT_EXIST

LINE_NOT_PRIMARY
    public static final int LINE_NOT_PRIMARY

LINECONTROL_FAILURE
    public static final int LINECONTROL_FAILURE

MAX_NUMBER_OF_CTI_CONNECTIONS_REACHED
    public static final int MAX_NUMBER_OF_CTI_CONNECTIONS_REACHED

MSGWAITING_DESTN_INVALID
    public static final int MSGWAITING_DESTN_INVALID

NO_ACTIVE_DEVICE_FOR_THIRDPARTY
    public static final int NO_ACTIVE_DEVICE_FOR_THIRDPARTY

NO_CONFERENCE_BRIDGE
    public static final int NO_CONFERENCE_BRIDGE

NOT_INITIALIZED
    public static final int NOT_INITIALIZED

PROTOCOL_TIMEOUT
    public static final int PROTOCOL_TIMEOUT

PROVIDER_ALREADY_OPEN
    public static final int PROVIDER_ALREADY_OPEN

PROVIDER_CLOSED
    public static final int PROVIDER_CLOSED

PROVIDER_NOT_OPEN
    public static final int PROVIDER_NOT_OPEN
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDIRECT_CALL_TABLE_FULL</td>
<td>Call Table Full</td>
</tr>
<tr>
<td>REDIRECT_CALL_DESTINATION_BUSY</td>
<td>Destination Busy</td>
</tr>
<tr>
<td>REDIRECT_CALL_DESTINATION_OUT_OF_ORDER</td>
<td>Destination Out of Order</td>
</tr>
<tr>
<td>REDIRECT_CALL_DIGIT_ANALYSIS_TIMEOUT</td>
<td>Digit Analysis Timeout</td>
</tr>
<tr>
<td>REDIRECT_CALL_DOES_NOT_EXIST</td>
<td>Does Not Exist</td>
</tr>
<tr>
<td>REDIRECT_CALL_INCOMPATIBLE_STATE</td>
<td>Incompatible State</td>
</tr>
<tr>
<td>REDIRECT_CALL_MEDIA_CONNECTION_FAILED</td>
<td>Media Connection Failed</td>
</tr>
<tr>
<td>REDIRECT_CALL_NORMAL_CLEARING</td>
<td>Normal Clearing</td>
</tr>
<tr>
<td>REDIRECT_CALL_ORIGINATOR_ABANDONED</td>
<td>Originator Abandoned</td>
</tr>
<tr>
<td>REDIRECT_CALL_PARTY_TABLE_FULL</td>
<td>Party Table Full</td>
</tr>
<tr>
<td>REDIRECT_CALL_PENDING_REDIRECT_TRANSACTION</td>
<td>Pending Redirect Transaction</td>
</tr>
<tr>
<td>REDIRECT_CALL_PROTOCOL_ERROR</td>
<td>Protocol Error</td>
</tr>
</tbody>
</table>
### Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDIRECT_CALL_UNKNOWN_DESTINATION</td>
<td>Redirect call unknown destination</td>
</tr>
<tr>
<td>REDIRECT_CALL_UNKNOWN_ERROR</td>
<td>Redirect call unknown error</td>
</tr>
<tr>
<td>REDIRECT_CALL_UNKNOWN_PARTY</td>
<td>Redirect call unknown party</td>
</tr>
<tr>
<td>REDIRECT_CALL_UNRECOGNIZED_MANAGER</td>
<td>Redirect call unrecognized manager</td>
</tr>
<tr>
<td>REDIRECT_CALLINFO_ERR</td>
<td>Redirect callinfo error</td>
</tr>
<tr>
<td>REDIRECT_ERR</td>
<td>Redirect error</td>
</tr>
<tr>
<td>RETRIEVEFAILED</td>
<td>Retrieve failed</td>
</tr>
<tr>
<td>RETRIEVEFAILED_ACTIVE_CALL_ON_LINE</td>
<td>Retrieve failed active call on line</td>
</tr>
<tr>
<td>SSAPI_NOT_REGISTERED</td>
<td>SSAPI not registered</td>
</tr>
<tr>
<td>TEMPORARY_FAILURE</td>
<td>Temporary failure</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>Timeout</td>
</tr>
<tr>
<td>TRANSFER_INACTIVE</td>
<td>Transfer inactive</td>
</tr>
</tbody>
</table>
TRANSFERFAILED
public static final int TRANSFERFAILED

TRANSFERFAILED_CALLCONTROL_TIMEOUT
public static final int TRANSFERFAILED_CALLCONTROL_TIMEOUT

TRANSFERFAILED_DESTINATION_BUSY
public static final int TRANSFERFAILED_DESTINATION_BUSY

TRANSFERFAILED_DESTINATION_UNALLOCATED
public static final int TRANSFERFAILED_DESTINATION_UNALLOCATED

TRANSFERFAILED_OUTSTANDING_TRANSFER
public static final int TRANSFERFAILED_OUTSTANDING_TRANSFER

UNDEFINED_LINE
public static final int UNDEFINED_LINE

UNKNOWN_GLOBAL_CALL_HANDLE
public static final int UNKNOWN_GLOBAL_CALL_HANDLE

UNRECOGNIZABLE_PDU
public static final int UNRECOGNIZABLE_PDU

UNSPECIFIED
public static final int UNSPECIFIED

The CTI error codes. These are the error codes that may be returned by CTI requests.

Methods

getErrorCode()
public int getErrorCode()

Returns the errorCode for this exception

Returns: errorCode in an integer representation
**getErrorDescription()**

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

This method returns the detail description of the errorCode

**Returns:** String detail description of the errorCode

**getErrorDescription(int)**

```java
public java.lang.String getErrorDescription(int errorCode)
```

*Deprecated.* instead use String getErrorDescription();

This method returns the detail description of the errorCode

**Returns:** String detail description of the errorCode

**getErrorName()**

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

This method returns an exception in the string format.

**Returns:** String representation of the error code

**getErrorName(int)**

```java
public java.lang.String getErrorName(int errorCode)
```

*Deprecated.* instead use String getErrorName();

This method will return an exception in the string format.

**Returns:** String representation of the error code

---

**CiscoJtapiPeer**

**Declaration**

```java
public interface CiscoJtapiPeer extends 
com.cisco.services.tracing.TraceModule, 
javax.telephony.JtapiPeer, CiscoObjectContainer
```
All Superinterfaces

CiscoObjectContainer, javax.telephony.JtapiPeer, com.cisco.services.tracing.TraceModule

Description

By extending the com.cisco.services.tracing.TraceModule interface, the CiscoJtapiPeer exposes trace information to applications. All instances of JtapiPeer objects created by the Cisco JTAPI implementation implement this interface. Applications that wish to manipulate the trace settings of the Cisco JTAPI implementation may use the CiscoJtapiPeer.getTraceManager method to obtain its TraceManager object. The TraceManager object may then be manipulated as described in the com.cisco.services.tracing package.

See Also:
com.cisco.services.tracing.TraceModule

Member Summary

Methods
CiscoJtapiProperties getJtapiProperties()
CiscoJtapiProperties defines the various methods that applications can use to modify the parameters that the JTAPI layer will use.

Inherited Member Summary

Methods inherited from interface CiscoObjectContainer
getObject(), setObject(Object)

Methods inherited from interface JtapiPeer
CiscoJTapiProperties defines the various methods that applications can use to modify the parameters that the JTAPI layer will use.

**See Also:** CiscoJTapiProperties

### Methods

**getJTapiProperties()**

```java
public com.cisco.jtapi.extensions.CiscoJTapiProperties
    getJTapiProperties()
```

CiscoJTapiProperties defines the various methods that applications can use to modify the parameters that the JTAPI layer will use.

**See Also:** CiscoJTapiProperties

### Description

Cisco JTAPI’s behaviour and functionality is tailored by many parameters which are read in from the jtapi.ini file when an instance of CiscoJTapiPeer is instantiated. These parameters are now exposed to applications for control via this CiscoJTapiProperties interface.

Applications can query the CiscoJTapiProperties properties object and change these parameters to better suit the application functionality. Exposing these properties via the CiscoJTapiProperties interface also allows
applications to have a single point of administration (at the application end) for these parameters.

The most visible parameters are those describing the tracing levels and tracing destinations.

**Usage:**

```java
JtapiPeer peer = JtapiPeerFactory.getJtapiPeer ( null );
if(peer instanceof CiscoJtapiPeer){
    CiscoJtapiProperties jProps = ((CiscoJtapiPeer)peer).getJtapiProperties();
    jProps.setTracePath("D:\Traces\WorkFlow");
    jProps.setUseJavaConsoleTrace(false);
    MyProviderObserver providerObserver = new MyProviderObserver ();
    provider = peer.getProvider ( providerName );
}
```

In the above example an application has set the java console tracing to off and set the trace path to D:\Traces\WorkFlowApp1.

When the peer is obtained an object implementing CiscoJtapiProperties is created by reading parameters set in the `jtapi.ini` file. If no `jtapi.ini` file exists in the classpath default settings are used to create this object. The parameters used by Cisco Jtapi are read in and frozen when the first `getProvider()` call is made.

### Member Summary

**Methods**

- `java.lang.String getAlarmServiceHostname()`  
  get the alarm service host name
- `int getAlarmServicePort()`  
  get the port number for the alarm service
- `int getCtiRequestTimeout()`  
  get the timeout for cti requests, other than the provider open (seconds)
- `java.lang.String[] getDebuggingNames()`  
  get names of supported debugging level jtapi traces
- `boolean getDebuggingValue(java.lang.String debuggingName)`  
  get the enabled or disabled state of a debugging level trace
### Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getDesiredServerHeartbeatInterval()</code></td>
<td>get the desired interval at which the CTI Manager must send heartbeats to JTAPI (seconds).</td>
</tr>
<tr>
<td><code>java.lang.String getFileNameBase()</code></td>
<td>the filename for individual log files.</td>
</tr>
<tr>
<td><code>java.lang.String getFileNameExtension()</code></td>
<td>get the filename extension for log files</td>
</tr>
<tr>
<td><code>int getNumTraceFiles()</code></td>
<td>number of trace files before rollover</td>
</tr>
<tr>
<td><code>boolean getPeriodicWakeupEnabled()</code></td>
<td>get the enabled state of periodic wake up</td>
</tr>
<tr>
<td><code>int getPeriodicWakeupInterval()</code></td>
<td>get the interval for periodic wakeup (milliseconds)</td>
</tr>
<tr>
<td><code>int getProviderOpenRequestTimeout()</code></td>
<td>get the timeout for a provider open request (seconds)</td>
</tr>
<tr>
<td><code>int getProviderRetryInterval()</code></td>
<td>get the interval at which the connection to the CTI Manager will be retried (seconds)</td>
</tr>
<tr>
<td><code>int getQueueSizeThreshold()</code></td>
<td>Get the threshold for the event queue size to trigger alarms</td>
</tr>
<tr>
<td><code>boolean getQueueStatsEnabled()</code></td>
<td>get the enabled state of event queue stats</td>
</tr>
<tr>
<td><code>int getRouteSelectTimeout()</code></td>
<td>get the route select timeout (milliseconds)</td>
</tr>
<tr>
<td><code>java.lang.String[] getServices()</code></td>
<td>Returns the services that this implementation supports.</td>
</tr>
<tr>
<td><code>java.lang.String getSyslogCollector()</code></td>
<td>get the syslog collector hostname</td>
</tr>
<tr>
<td><code>int getSyslogCollectorUDPPort()</code></td>
<td>get the syslog collector UDP port</td>
</tr>
<tr>
<td><code>java.lang.String getTraceDirectory()</code></td>
<td>The directory in the path where trace files will be written</td>
</tr>
<tr>
<td><code>int getTraceFileSize()</code></td>
<td>The size of the tracefiles before rollover</td>
</tr>
<tr>
<td><code>java.lang.String[] getTraceNames()</code></td>
<td>get the names of supported jtapi traces</td>
</tr>
<tr>
<td><code>java.lang.String getTracePath()</code></td>
<td>get the path where the trace files will be located</td>
</tr>
<tr>
<td><code>boolean getTraceValue(java.lang.String traceName)</code></td>
<td>get the enabled or disabled state of a trace</td>
</tr>
</tbody>
</table>
## Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean getUseAlarmService()</td>
<td>get the enabled/disabled state of the alarm service</td>
</tr>
<tr>
<td>boolean getUseFileTrace()</td>
<td>get the enabled or disabled state of jtapi log file tracing</td>
</tr>
<tr>
<td>boolean getUseJavaConsoleTrace()</td>
<td>get the enabled or disabled state of jtapi console tracing</td>
</tr>
<tr>
<td>boolean getUseSameDir()</td>
<td>if UseSameDir is true this will cause the traces to go to a single directory.</td>
</tr>
<tr>
<td>boolean getUseSyslog()</td>
<td>get the enabled or disabled state of syslog tracing</td>
</tr>
<tr>
<td>void setAlarmServiceHostname(java.lang.String hostname)</td>
<td>set the alarm service host name</td>
</tr>
<tr>
<td>void setAlarmServicePort(int portNumber)</td>
<td>set the port number the alarm service is listening on</td>
</tr>
<tr>
<td>void setCtiRequestTimeout(int seconds)</td>
<td>set the timeout for cti requests other than provider open (seconds)</td>
</tr>
<tr>
<td>void setDebuggingValue(java.lang.String debuggingName, boolean value)</td>
<td>enable or disable a particular debugging level trace</td>
</tr>
<tr>
<td>void setDesiredServerHeartbeatInterval(int seconds)</td>
<td>set the desired interval at which the CTI Manager must send heartbeats to JTAPI (seconds).</td>
</tr>
<tr>
<td>void setFileNameBase(java.lang.String base)</td>
<td>set the filename for log files</td>
</tr>
<tr>
<td>void setFileNameExtension(java.lang.String extn)</td>
<td>set the filename extension for log files</td>
</tr>
<tr>
<td>void setNumTraceFiles(int val)</td>
<td>set the number of trace files before rollover</td>
</tr>
<tr>
<td>void setPeriodicWakeupEnabled(boolean enabled)</td>
<td>set the enable/disable state for periodic wake up</td>
</tr>
<tr>
<td>void setPeriodicWakeupInterval(int milliseconds)</td>
<td>set the periodic wake up interval (milliseconds)</td>
</tr>
<tr>
<td>void setProviderOpenRequestTimeout(int seconds)</td>
<td>set the timeout for a provider open request (seconds)</td>
</tr>
<tr>
<td>void setProviderRetryInterval(int seconds)</td>
<td>set the interval at which the connection to the CTI Manager will ge retried (seconds)</td>
</tr>
<tr>
<td>void setQueueSizeThreshold(int size)</td>
<td>Set the threshold for the event queue size to trigger alarms</td>
</tr>
<tr>
<td>void setQueueStatsEnabled(boolean enabled)</td>
<td>enable / disable event queue statistics</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void setRouteSelectTimeout(int milliseconds)</code></td>
<td>set the route select timeout milliseconds</td>
</tr>
<tr>
<td><code>void setServices(java.lang.String services)</code></td>
<td>set a list of available services</td>
</tr>
<tr>
<td><code>void setSyslogCollector(java.lang.String value)</code></td>
<td>set the syslog collector hostname</td>
</tr>
<tr>
<td><code>void setSyslogCollectorUDPPort(int port)</code></td>
<td>set the syslog collector UDP port</td>
</tr>
<tr>
<td><code>void setTraceDirectory(java.lang.String dir)</code></td>
<td>set the directory where jtapi trace files should be written</td>
</tr>
<tr>
<td><code>void setTraceFileSize(int val)</code></td>
<td>Set the size of the trace file</td>
</tr>
<tr>
<td><code>void setTracePath(java.lang.String path)</code></td>
<td>set the directory root where jtapi traces will be written</td>
</tr>
<tr>
<td><code>void setTraceValue(java.lang.String traceName, boolean value)</code></td>
<td>enable or disable a particular trace</td>
</tr>
<tr>
<td><code>void setUseAlarmService(boolean value)</code></td>
<td>enable the alarm service</td>
</tr>
<tr>
<td><code>void setUseFileTrace(boolean value)</code></td>
<td>enable or disable jtapi log file tracing</td>
</tr>
<tr>
<td><code>void setUseJavaConsoleTrace(boolean value)</code></td>
<td>enable or disable jtapi console tracing</td>
</tr>
<tr>
<td><code>void setUseSameDi</code></td>
<td>if UseSameDir is true this will cause the traces to go to a single directory.</td>
</tr>
<tr>
<td><code>void setUseSyslog(boolean value)</code></td>
<td>enable or disable syslog tracing</td>
</tr>
</tbody>
</table>

Methods

getAlarmServiceHostname()

public java.lang.String getAlarmServiceHostname()

get the alarm service host name

getAlarmServicePort()

public int getAlarmServicePort()

get the port number for the alarm service
getCtiRequestTimeout()

public int getCtiRequestTimeout()

get the timeout for cti requests, other than the provider open (seconds)

getDebuggingNames()

public java.lang.String[] getDebuggingNames()

get names of supported debugging level jtapi traces

getDebuggingValue(String)

public boolean getDebuggingValue(java.lang.String debuggingName)

get the enabled or disabled state of a debugging level trace

getDesiredServerHeartbeatInterval()

public int getDesiredServerHeartbeatInterval()

get the desired interval at which the CTI Manager must send heartbeats to JTAPI (seconds). The actual interval is decided by the server at connect time.

getFileNameBase()

public java.lang.String getFileNameBase()

the filename for individual log files.

getFileNameExtension()

public java.lang.String getFileNameExtension()

get the filename extension for log files

getNumTraceFiles()

public int getNumTraceFiles()

number of trace files before rollover

getPeriodicWakeupEnabled()

public boolean getPeriodicWakeupEnabled()

get the enabled state of periodic wake up
getPeriodicWakeupInterval()

public int getPeriodicWakeupInterval()

get the interval for periodic wakeup (milliseconds)

getProviderOpenRequestTimeout()

public int getProviderOpenRequestTimeout()

get the timeout for a provider open request (seconds)

getProviderRetryInterval()

public int getProviderRetryInterval()

get the interval at which the connection to the CTI Manager will be retried (seconds)

getQueueSizeThreshold()

public int getQueueSizeThreshold()

Get the threshold for the event queue size to trigger alarms

getQueueStatsEnabled()

public boolean getQueueStatsEnabled()

get the enabled state of event queue stats

getRouteSelectTimeout()

public int getRouteSelectTimeout()

get the route select timeout (milliseconds)

getServices()

public java.lang.String[] getServices()

Returns the services that this implementation supports. Note: This is a static list administered in the jtapi.ini file. There is no automatic discovery mechanism to locate available cti services

getSyslogCollector()

public java.lang.String getSyslogCollector()

get the syslog collector hostname
getSyslogCollectorUDPPort()
public int getSyslogCollectorUDPPort()
get the syslog collector UDP port

getTraceDirectory()
public java.lang.String getTraceDirectory()
The directory in the path where trace files will be written

getTraceFileSize()
public int getTraceFileSize()
The size of the tracefiles before rollover

getTraceNames()
public java.lang.String[] getTraceNames()
get the names of supported jitapi traces

getTracePath()
public java.lang.String getTracePath()
get the path where the trace files will be located

getTraceValue(String)
public boolean getTraceValue(java.lang.String traceName)
get the enabled or disabled state of a trace

getUseAlarmService()
public boolean getUseAlarmService()
get the enabled/disabled state of the alarm service

getUseFileTrace()
public boolean getUseFileTrace()
get the enabled or disabled state of jitapi log file tracing

getUseJavaConsoleTrace()
public boolean getUseJavaConsoleTrace()
get the enabled or disabled state of jtapi console tracing

```java
public boolean getUseSameDir()
```

if UseSameDir is true this will cause the traces to go to a single directory. Otherwise each instance of a jtapi application will cause the traces to go to a separate directory, indexed in sequence from the last directory written or available.

```java
public boolean getUseSyslog()
```

get the enabled or disabled state of syslog tracing

```java
public void setAlarmServiceHostname(java.lang.String hostname)
```

set the alarm service host name

```java
public void setAlarmServicePort(int portNumber)
```

set the port number the alarm service is listening on

```java
public void setCtiRequestTimeout(int seconds)
```

set the timeout for cti requests other than provider open (seconds)

```java
public void setDebuggingValue(java.lang.String debuggingName, boolean value)
```

enable or disable a particular debugging level trace

```java
public void setDesiredServerHeartbeatInterval(int seconds)
```

set the desired interval at which the CTI Manager must send heartbeats to JTAPI (seconds). The actual interval is decided by the server at connect time.
setFileNameBase(String)
   public void setFileNameBase(String base)
   set the filename for log files

setFileNameExtension(String)
   public void setFileNameExtension(String extn)
   set the filename extension for log files

setNumTraceFiles(int)
   public void setNumTraceFiles(int val)
   set the number of trace files before rollover

setPeriodicWakeupEnabled(boolean)
   public void setPeriodicWakeupEnabled(boolean enabled)
   set the enable/disable state for periodic wake up

setPeriodicWakeupInterval(int)
   public void setPeriodicWakeupInterval(int milliseconds)
   set the periodic wake up interval (milliseconds)

setProviderOpenRequestTimeout(int)
   public void setProviderOpenRequestTimeout(int seconds)
   set the timeout for a provider open request (seconds)

setProviderRetryInterval(int)
   public void setProviderRetryInterval(int seconds)
   set the interval at which the connection to the CTI Manager will ge retried (seconds)

setQueueSizeThreshold(int)
   public void setQueueSizeThreshold(int size)
   Set the threshold for the event queue size to trigger alarms
setQueueStatsEnabled(boolean)
    public void setQueueStatsEnabled(boolean enabled)
    enable / disable event queue statistics

setRouteSelectTimeout(int)
    public void setRouteSelectTimeout(int milliseconds)
    set the route select timeout milliseconds

setServices(String[])
    public void setServices(java.lang.String[] services)
    set a list of available services

setSyslogCollector(String)
    public void setSyslogCollector(java.lang.String value)
    set the syslog collector hostname

setSyslogCollectorUDPPort(int)
    public void setSyslogCollectorUDPPort(int port)
    set the syslog collector UDP port

setTraceDirectory(String)
    public void setTraceDirectory(java.lang.String dir)
    Set the directory where jtapi trace files should be written

setTraceFileSize(int)
    public void setTraceFileSize(int val)
    Set the size of the trace file

setTracePath(String)
    public void setTracePath(java.lang.String path)
    set the directory root where jtapi traces will be written
setTraceValue(String, boolean)

public void setTraceValue(java.lang.String traceName, 
boolean value)

enable or disable a particular trace

setUseAlarmService(boolean)

public void setUseAlarmService(boolean value)

enable the alarm service

setUseFileTrace(boolean)

public void setUseFileTrace(boolean value)

enable or disable jtapi log file tracing

setUseJavaConsoleTrace(boolean)

public void setUseJavaConsoleTrace(boolean value)

enable or disable jtapi console tracing

setUseSameDir(boolean)

public void setUseSameDir(boolean value)

if UseSameDir is true this will cause the traces to go to a single directory. Otherwise each instance of a jtapi application will cause the traces to go to a separate directory, indexed in sequence from the last directory written or available.

setUseSyslog(boolean)

public void setUseSyslog(boolean value)

enable or disable syslog tracing
CiscoJ tapiVersion

Declaration

```java
public class CiscoJtapiVersion
    java.lang.Object
    |  +-- com.cisco.jtapi.extensions.CiscoJtapiVersion
```

Description

This class gives the version information of the installed Cisco JTAPI. Programs can get the version number using the accessor methods. Cisco Jtapi Version is in a.b(x.y) format where a indicates the major version b indicates the minor version x indicates the revision number y indicates the build number

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructors</strong></td>
</tr>
<tr>
<td>CiscoJtapiVersion()</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>java.lang.String getBuildDescription()</td>
</tr>
<tr>
<td>Returns 'release' if it is a release version or debug if it is not a release version</td>
</tr>
<tr>
<td>int getBuildNumber()</td>
</tr>
<tr>
<td>This returns the build number of the version</td>
</tr>
<tr>
<td>int getMajorVersion()</td>
</tr>
<tr>
<td>This method returns the major version number</td>
</tr>
<tr>
<td>int getMinorVersion()</td>
</tr>
<tr>
<td>This method returns the minor version number</td>
</tr>
<tr>
<td>int getRevisionNumber()</td>
</tr>
<tr>
<td>This method returns the revision number of the version</td>
</tr>
<tr>
<td>java.lang.String getVersion()</td>
</tr>
<tr>
<td>returns the version information in a.b(x.y) format without name</td>
</tr>
</tbody>
</table>
Constructors

CiscoJtapiVersion()
    public CiscoJtapiVersion()

Methods

getBuildDescription()
    public java.lang.String getBuildDescription()
    Returns 'release' if it is a release version or debug if it is not a release version

getBuildNumber()
    public int getBuildNumber()
    This returns the build number of the vesion

getMajorVersion()
    public int getMajorVersion()
    This method returns the major version number

toString()
    java.lang.String toString()
    returns the version information in a.b(x.y) format
getMinorVersion()

public int getMinorVersion()

This method returns the minor version number

getRevisionNumber()

public int getRevisionNumber()

This method returns the revision number of the version

getVersion()

public java.lang.String get\n
returns the version information in a.b(x.y) format without name

toString()

public java.lang.String toString()

returns the version information in a.b(x.y) format

Overrides: toString in class Object

CiscoMediaCapability

Declaration

public class CiscoMediaCapability

java.lang.Object

|--com.cisco.jtapi.extensions.CiscoMediaCapability

Direct Known Subclasses

CiscoG711MediaCapability, CiscoG723MediaCapability, CiscoG729MediaCapability, CiscoGSMMediaCapability
CiscoMediaCapability

Description

The CiscoMediaCapability object specifies the properties of a particular media format that an application can support for CiscoMediaTerminals that it registers. Because CiscoMediaCapability is an abstract class, applications may only construct its subclasses directly.

See Also:
CiscoG711MediaCapability, CiscoG723MediaCapability, CiscoG729MediaCapability, CiscoGSMMediaCapability, CiscoRTPBitRate, CiscoRTPPayload

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static CiscoMediaCapability G711_64K_30_MILLISECONDS</td>
</tr>
<tr>
<td>static CiscoMediaCapability G723_6K_30_MILLISECONDS</td>
</tr>
<tr>
<td>static CiscoMediaCapability G729_30_MILLISECONDS</td>
</tr>
<tr>
<td>static CiscoMediaCapability GSM_80_MILLISECONDS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoMediaCapability(int payloadType, int maxFramesPerPacket)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getMaxFramesPerPacket()</td>
</tr>
<tr>
<td>int getPayloadType()</td>
</tr>
<tr>
<td>java.lang.String toString()</td>
</tr>
</tbody>
</table>
Fields

**G711_64K_30_MILLISECONDS**

```java
public static final com.cisco.jtapi.extensions.CiscoMediaCapability G711_64K_30_MILLISECONDS
```

G.711 capability with default parameters

See Also:

CiscoG711MediaCapability

**G723_6K_30_MILLISECONDS**

```java
public static final com.cisco.jtapi.extensions.CiscoMediaCapability G723_6K_30_MILLISECONDS
```

G.723 capability with default parameters

See Also:

CiscoG723MediaCapability

**G729_30_MILLISECONDS**

```java
public static final com.cisco.jtapi.extensions.CiscoMediaCapability G729_30_MILLISECONDS
```

G.729 capability with default parameters

See Also:

CiscoG729MediaCapability
CiscoMediaCapability

GSM_80_MILLISECONDS

public static final com.cisco.jtapi.extensions.CiscoMediaCapability GSM_80_MILLISECONDS

GSM capability with default parameters

See Also:
CiscoGSMMediaCapability

Constructors

CiscoMediaCapability(int, int)

public CiscoMediaCapability(int payloadType, int maxFramesPerPacket)

Constructs a CiscoMediaCapability object for the specified payload type and packet size.

Methods

getMaxFramesPerPacket()

public int getMaxFramesPerPacket()

Returns the packet size specified by this object.

Returns: the packet size, specified as a the number of frames within a single packet

getPayloadType()

public int getPayloadType()

Returns the payload type specified by this object.

Returns: a payload type from the RTPPayload interface
public interface CiscoMediaTerminal extends CiscoTerminal, javax.telephony.Terminal

A CiscoMediaTerminal is a special kind of CiscoTerminal that allows applications to terminate RTP media streams. Unlike a CiscoTerminal, a CiscoMediaTerminal does not represent a physical telephony endpoint, which is observable and controllable in a third-party manner. Instead, a CiscoMediaTerminal is a logical telephony endpoint, which may be associated with any application that desires to terminate media. Such applications include voice mail systems, interactive voice response (IVR), and “soft” phones.

Note

**Only CTIPorts appear as CiscoMediaTerminals through JTAPI.**

Terminating media is a two-step process. To terminate media for a particular terminal, an application adds an observer that implements the CiscoTerminalObserver interface using the Terminal.addObserver method. Finally, the application registers its IP
address and port number to which the Terminal’s incoming RTP streams are to be directed using the CiscoMediaTerminal.register method.

See Also:
CiscoTerminal

### Member Summary

#### Methods

```java
void changeRTPDefaults(java.net.InetAddress address, int port)
Changes the default registration parameters to specified address and port.
```

```java
boolean isRegistered()
This method returns true if the CiscoMediaTerminal is registered and false otherwise.
```

```java
void register(java.net.InetAddress address, int port, CiscoMediaCapability capabilities)
The CiscoMediaTerminal must be in the CiscoTerminal.UNREGISTERED state and its Provider must be in the Provider.IN_SERVICE state.
```

```java
void register(java.net.InetAddress address, int port, CiscoMediaCapability capabilities)
The CiscoMediaTerminal must be in the CiscoTerminal.UNREGISTERED state and its Provider must be in the Provider.IN_SERVICE state.
```

```java
void unregister()
The CiscoMediaTerminal must not be registered and its Provider must be in the Provider.IN_SERVICE state.
```

### Inherited Member Summary

#### Fields inherited from interface CiscoTerminal

- IN_SERVICE, OUT_OF_SERVICE

#### Methods inherited from interface CiscoObjectContainer

- `getObject()`, `setObject(Object)`

#### Methods inherited from interface CiscoTerminal
Chapter 2      Cisco JTAPI Implementation

CiscoMediaTerminal

**Methods**

**changeRTPDefaults(InetAddress, int)**

```java
public void changeRTPDefaults(java.net.InetAddress address,
                               int port)
                        throws CiscoRegistrationException
```

Changes the default registration parameters to specified address and port. Only Registered application may invoke this method.

**Parameters:**
- `address` - the internet address for inbound RTP streams on this terminal
- `port` - the UDP port for inbound RTP streams on this terminal

**Throws:**
- `CiscoRegistrationException`

**isRegistered()**

```java
public boolean isRegistered()
```

This method returns true if the CiscoMediaTerminal is registered and false otherwise.

**register(InetAddress, int)**

```java
public void register(java.net.InetAddress address, int port)
                        throws CiscoRegistrationException
```

**Methods inherited from interface `Terminal`**

- `addCallObserver(CallObserver)`, `addObserver(TerminalObserver)`, `getAddresses()`, `getCallObservers()`, `getCapabilities()`, `getName()`, `getObservers()`, `getProvider()`, `getTerminalCapabilities(Terminal, Address)`, `getTerminalConnections()`, `removeCallObserver(CallObserver)`, `removeObserver(TerminalObserver)`
**Deprecated.** Registers a Terminal with the specified address and port, defaulting to G.711 64KHz u-law encoding with a thirty millisecond packet size.

**Parameters:**
- `address` - the internet address for inbound RTP streams on this terminal
- `port` - the UDP port for inbound RTP streams on this terminal

**Throws:**
- `CiscoRegistrationException`

```java
public void register(InetAddress address, int port, CiscoMediaCapability[] capabilities)
throws CiscoRegistrationException
```

The CiscoMediaTerminal must be in the `CiscoTerminal.UNREGISTERED` state and its Provider must be in the `Provider.IN_SERVICE` state. The successful effect of this method is to register the MediaTerminal.

**Method Arguments**
This method has three arguments. The first argument specifies the internet address at which the RTP media stream for this Terminal will be terminated, the second indicates the UDP port at which RTP packets will be directed, and the final argument indicates the type of RTP encodings that the application is willing to support for this Terminal.

**Method Post-conditions**
This method returns successfully when the MediaTerminal is registered.

**Parameters:**
- `address` - the internet address for inbound RTP streams on this terminal
- `port` - the UDP port for inbound RTP streams on this terminal
- `capabilities` - the list of RTP encodings supported by this terminal

**Throws:**
- `CiscoRegistrationException`
unregister()  
public void **unregister**()  
throws CiscoUnregistrationException  

The CiscoMediaTerminal must not be registered and its Provider must be in the Provider.IN_SERVICE state. The successful effect of this method is to unregister the MediaTerminal.

**Method Post-conditions**
This method returns successfully when the MediaTerminal is unregistered.

**Throws:**  
CiscoUnregistrationException

---

**CiscoObjectContainer**

**Declaration**

public interface **CiscoObjectContainer**

**All Known Subinterfaces**

CiscoAddress, CiscoCall, CiscoCallID, CiscoConnection, CiscoConnectionID, CiscoConsultCall, CiscoJtapiPeer, CiscoMediaTerminal, CiscoProvider, CiscoTerminal, CiscoTerminalConnection

**Description**

The ApplicationObject interface allows applications to associate an application defined object to objects that implement this interface.
Member Summary

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Object getObject()</code></td>
<td>Gets the application-defined object.</td>
</tr>
</tbody>
</table>

Methods

- `getObject()`
  - public `java.lang.Object` `getObject()`
  - Gets the application-defined object.
  - **Returns:** the CallID property of this Call

- `setObject(Object)`
  - public `java.lang.Object` `setObject(java.lang.Object reference)`
  - Sets an application-defined object.
  - **Returns:** the CallManagerID property of this Call

CiscoOutOfServiceEv

Declaration

```java
public interface CiscoOutOfServiceEv extends CiscoEv
```

All Superinterfaces

- `CiscoEv`, `javax.telephony.events.Ev`
All Known Subinterfaces

CiscoAddrOutOfServiceEv, CiscoTermOutOfServiceEv

Description

The CiscoAddrOutOfServiceEv event

Member Summary

Fields

- static int CAUSE_CALLMANAGER_FAILURE
- static int CAUSE_CTIMANAGER_FAILURE
- static int CAUSE_DEVICE_FAILURE
- static int CAUSE_DEVICE_UNREGISTERED
- static int CAUSE_NOCALLMANAGER_AVAILABLE
- static int CAUSE_REHOMING_FAILURE
- static int ID

Inherited Member Summary

Fields inherited from interface Ev

- CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
  CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
  CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
  CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
  META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
  META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

- getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()
Fields

**CAUSE_CALLMANAGER_FAILURE**
public static final int CAUSE_CALLMANAGER_FAILURE

**CAUSE_CTIMANAGER_FAILURE**
public static final int CAUSE_CTIMANAGER_FAILURE

**CAUSE_DEVICE_FAILURE**
public static final int CAUSE_DEVICE_FAILURE

**CAUSE_DEVICE_UNREGISTERED**
public static final int CAUSE_DEVICE_UNREGISTERED

**CAUSE_NOCALLMANAGER_AVAILABLE**
public static final int CAUSE_NOCALLMANAGER_AVAILABLE

**CAUSE_REHOME_TO_HIGHER_PRIORITY_CM**
public static final int CAUSE_REHOME_TO_HIGHER_PRIORITY_CM

**CAUSE_REHOMING_FAILURE**
public static final int CAUSE_REHOMING_FAILURE

**ID**
public static final int ID

CiscoProvCallParkEv

Declaration

public interface CiscoProvCallParkEv extends CiscoProvFeatureEv
All Superinterfaces

CiscoEv, CiscoProvEv, CiscoProvFeatureEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv

Description

Introduction
The CiscoTermSoftKeyPressedEv event

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Indicates that a call is parked</td>
</tr>
<tr>
<td>PARK_STATE_ACTIVE</td>
<td>Indicates that a call is inactive</td>
</tr>
<tr>
<td>PARK_STATE_IDLE</td>
<td>Indicates that a call is idle</td>
</tr>
<tr>
<td>REASON_CALLPARK</td>
<td>This event is due to call park</td>
</tr>
<tr>
<td>REASON_CALLPARKREMAINDER</td>
<td>This event is due to call park remainder</td>
</tr>
<tr>
<td>REASON_CALLUNPARK</td>
<td>This event is due to call being unparked</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getintCallIDValue()</td>
<td>Returns an integer representation of this object</td>
</tr>
<tr>
<td>getParkDN()</td>
<td>This returns where the call is parked</td>
</tr>
<tr>
<td>getReason()</td>
<td>This returns the reason of the event</td>
</tr>
<tr>
<td>getParkedParty()</td>
<td>This returns the DN of the parked party</td>
</tr>
<tr>
<td>getParkingParty()</td>
<td>This returns the DN of the parking party</td>
</tr>
</tbody>
</table>
## Member Summary

```java
int getState()
```

This returns the state of the call. Possible states are:
- CiscoProvCallParkEv.PARK_STATE_IDLE
- CiscoProvCallParkEv.PARK_STATE_ACTIVE

## Inherited Member Summary

### Fields inherited from interface `Ev`

- `CAUSE_CALL_CANCELLED`, `CAUSE_DEST_NOT_OBTAINABLE`, `CAUSE_INCOMPATIBLE_DESTINATION`, `CAUSE_LOCKOUT`, `CAUSE_NETWORK_CONGESTION`, `CAUSE_NETWORK_NOT_OBTAINABLE`, `CAUSE_NEW_CALL`, `CAUSE_NORMAL`, `CAUSE_RESOURCES_NOT_AVAILABLE`, `CAUSE_SNAPSHOT`, `CAUSE_UNKNOWN`, `META_CALL_ADDITIONAL_PARTY`, `META_CALL_ENDING`, `META_CALL_MERGING`, `META_CALL_PROGRESS`, `META_CALL_REMOVING_PARTY`, `META_CALL_STARTING`, `META_CALL_TRANSFERRING`, `META_SNAPSHOT`, `META_UNKNOWN`

### Methods inherited from interface `CiscoProvFeatureEv`

- `getFeatureID()`

### Methods inherited from interface `Ev`

- `getCause()`, `getID()`, `getMetaCode()`, `getObserved()`, `isNewMetaEvent()`

### Methods inherited from interface `ProvEv`

- `getProvider()`

## Fields

### `ID`

```java
public static final int ID
```

### `PARK_STATE_ACTIVE`

```java
public static final int PARK_STATE_ACTIVE
```
Indicates that a call is parked

PARK_STATE_IDLE
   public static final int PARK_STATE_IDLE
   Indicates that a call is unparked

REASON_CALLPARK
   public static final int REASON_CALLPARK
   This event is due to call park

REASON_CALLPARKREMAINDE
   public static final int REASON_CALLPARKREMAINDE
   This event is due to call park remainder

REASON_CALLUNPARK
   public static final int REASON_CALLUNPARK
   This event is due to call being unparked

Methods

getintCallIDValue()
   public int getintCallIDValue()
   Returns an integer representation of this object

getParkDN()
   public java.lang.String getParkDN()
   This returns where the call is parked

getParkedParty()
   public java.lang.String getParkedParty()
   This returns the DN of the parked party
getParkingParty()

public java.lang.String getParkingParty()

This returns the DN of the parking party

getReason()

public int getReason()

This returns the reason of the event. Possible states are:
CiscoProvCallParkEv.REASON_CALLPARK
CiscoProvCallParkEv.REASON_CALLUNPARK
CiscoProvCallParkEv.REASON_CALLPARKREMAINDER

ggetState()

public int getState()

This returns the state of the call Possible states are
CiscoProvCallParkEv.PARK_STATE_IDLE
CiscoProvCallParkEv.PARK_STATE_ACTIVE

CiscoProvEv

Declaration

public interface CiscoProvEv extends CiscoEv,
javax.telephony.events.ProvEv

All Superinterfaces

CiscoEv, javax.telephony.events.Ev,
javax.telephony.events.ProvEv
All Known Subinterfaces

CiscoAddrCreatedEv, CiscoAddrRemovedEv, CiscoProvCallParkEv, CiscoProvFeatureEv, CiscoProvFeatureUnRegisteredEv, CiscoTermCreatedEv, CiscoTermRemovedEv

Description

The CiscoProvEv interface, which extends JTAPI’s core javax.telephony.eventsProvEv interface, serves as the base interface for all Cisco-extended JTAPI Provider events. Every Provider-related event in this package extends this interface, directly or indirectly.

See Also:
javax.telephony.eventsProvEv

Inherited Member Summary

Fields inherited from interface Ev
CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev
getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface ProvEv
getProvider()
CiscoProvFeatureEv

Declaration

public interface CiscoProvFeatureEv extends CiscoProvEv

All Superinterfaces

CiscoEv, CiscoProvEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv

All Known Subinterfaces

CiscoProvCallParkEv

Member Summary

Methods

int getFeatureID()
Feature ID for which application is interested in receiving events

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERING, META_SNAPSHOT, META_UNKNOWN
Inherited Member Summary

Methods inherited from interface `Ev`:
- `getCause()`, `getID()`, `getMetaCode()`, `getObserved()`, `isNewMetaEvent()`.

Methods inherited from interface `ProvEv`:
- `getProvider()`.

Methods

```java
public int getFeatureID()
```

Feature ID for which application is interested in receiving events.

CiscoProvFeatureID

**Declaration**

```java
public interface CiscoProvFeatureID
```

**Fields**

```java
static int MONITOR_CALLPARK_DN
```

Inherited Member Summary

**Member Summary**

**Fields**

```java
static int MONITOR_CALLPARK_DN
```

**Methods inherited from interface `Ev`**
- `getCause()`, `getID()`, `getMetaCode()`, `getObserved()`, `isNewMetaEvent()`.

**Methods inherited from interface `ProvEv`**
- `getProvider()`.

**Methods**

```java
public int getFeatureID()
```

Feature ID for which application is interested in receiving events.
CiscoProvFeatureUnRegisteredEv

Declaration

```
public interface CiscoProvFeatureUnRegisteredEv extends CiscoProvEv
```

All Superinterfaces

- CiscoEv, CiscoProvEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv

Description

The `CiscoProvFeatureUnRegisteredEv` event indicates the unregistration of a particular feature by Cisco CallManager.

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td>static int ID</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>int getFeatureID()</td>
</tr>
<tr>
<td>FeatureID for which application will no longer receive events</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Fields inherited from interface Ev
Inherited Member Summary

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev
getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface ProvEv
getProvider()

Fields

ID
public static final int ID

Methods

getFeatureID()
public int getFeatureID()

FeatureID for which application will no longer receive events

CiscoProvider

Declaration

public interface CiscoProvider extends javax.telephony.Provider,
CiscoObjectContainer
# All Superinterfaces

CiscoObjectContainer, javax.telephony.Provider

## Description

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
</table>
| void deleteCall(javax.telephony.Call call)  
Deletes an un used call created by createCall(). |
| boolean getCallbackGuardEnabled()  
Returns the current state of the callback guard feature |
| CiscoMediaTerminal getMediaTerminal(java.lang.String name)  
Returns an instance of the CiscoMediaTerminal class which corresponds to the given name. |
| CiscoMediaTerminal[] getMediaTerminals()  
Returns an array of CiscoMediaTerminals associated with the Provider and within the Provider’s domain. |
| java.lang.String getVersion()  
Returns the current version of provider running if provider is inservice otherwise it will return empty string |
| void registerFeature(int featureID)  
used to register for a particular feature for which application will get Provider events. |
| void setCallbackGuardEnabled(boolean enabled)  
Enables or disables try/catch logic for observer callbacks |
| void unregisterFeature(int featureID)  
used to unregister a particular feature. |

## Inherited Member Summary

### Fields inherited from interface Provider

IN_SERVICE, OUT_OF_SERVICE, SHUTDOWN
Inherited Member Summary

Methods inherited from interface CiscoObjectContainer

getObject(), setObject(Object)

Methods inherited from interface Provider

addObserver(ProviderObserver), createCall(), getAddress(String),
getAddressCapabilities(Terminal), getAddressCapabilities(Terminal), getAddresses(),
getCallCapabilities(Terminal, Address), getCallCapabilities(Terminal, Address),
getCalls(), getCapabilities(), getConnectionCapabilities(Terminal, Address),
getConnectionCapabilities(Terminal, Address), getName(), getObservers(),
getProviderCapabilities(Terminal), getProviderCapabilities(Terminal), getState(),
getTerminal(String), getTerminalCapabilities(Terminal),
getTerminalCapabilities(Terminal), getTerminalConnectionCapabilities(Terminal),
getTerminalConnectionCapabilities(Terminal), getTerminals(),
removeObserver(ProviderObserver), shutdown()

Methods

deleteCall(Call)

public void deleteCall(javax.telephony.Call call)
    throws InvalidStateException

Deletes an un used call created by createCall(). An exception is generated if
the call is not in IDLE state or if provider is not in Provider.IN_SERVICE
state. Applications may use this interface to move un used calls to INVALID
state and reclaim resources alloacated to the call.

Pre-conditions:
1. this.getState() == Provider.IN_SERVICE
2. call.getState() == Call.IDLE

Post-conditions:
1. call.getState() == Call.INVALID

Throws:
    javax.telephony.InvalidStateException
getCallbackGuardEnabled()

public boolean getCallbackGuardEnabled()

Returns the current state of the callback guard feature

Returns: the current state of the callback guard feature

getMediaTerminal(String)

public com.cisco.jtapi.extensions.CiscoMediaTerminal getMediaTerminal(java.lang.String name)

throws InvalidArgumentException

Returns an instance of the CiscoMediaTerminal class which corresponds to
the given name. Each CiscoMediaTerminal has a unique name associated
with it, which is assigned to it by the JTAPI implementation. If no
CiscoMediaTerminal is available for the given name within the Provider’s
domain, this method throws the InvalidArgumentException. This
CiscoMediaTerminal is contained in the arrays generated by
Provider.getTerminals() and CiscoProvider.getMediaTerminals().

Pre-conditions:

1. Let CiscoMediaTerminal terminal = this.getMediaTerminal(name);
2. terminal is an element of this.getTerminals();
3. terminal is an element of this.getMediaTerminals();

Post-conditions:

1. Let CiscoMediaTerminal terminal = this.getMediaTerminal(name);
2. terminal is an element of this.getTerminals();
3. terminal is an element of this.getMediaTerminals();

Parameters:

name - The name of desired CiscoMediaTerminal object.

Returns: The CiscoMediaTerminal object associated with the given name.

Throws:

javax.telephony.InvalidArgumentException - The name
provided does not correspond to a name of any CiscoMediaTerminal
known to the Provider or within the Provider’s domain.
getMediaTerminals()

```java
public com.cisco.jtapi.extensions.CiscoMediaTerminal[] getMediaTerminals()
    throws ResourceUnavailableException
```

Returns an array of CiscoMediaTerminals associated with the Provider and within the Provider’s domain. Each CiscoMediaTerminal possesses a unique name, which is assigned to it by the JTAPI implementation. If there are no CiscoMediaTerminals associated with this Provider, then this method returns null. This array is a subset of the array returned by Provider.getTerminals().

**Post-conditions:**

1. Let CiscoMediaTerminal[] terminals = this.getMediaTerminals()
2. terminals == null or terminals.length >= 1
3. if terminals != null, terminals is a subset of this.getTerminals ()

**Returns:** An array of Terminals in the Provider’s local domain.

**Throws:**

- `javax.telephony.ResourceUnavailableException` - Indicates the number of media terminals present in the Provider is too great to return as a static array.

getVersion()

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

Returns the current version of provider running if provider is inservice otherwise it will return empty string

registerFeature(int)

```java
public void registerFeature(int featureID)
    throws InvalidStateException, PrivilegeViolationException, InvalidArgumentException
```

used to register for a particular feature for which application will get Provider events. Applications should pass in the featureID of the softkey. Current supported features are listed in CiscoProvFeatureID interface
setCallbackGuardEnabled(boolean)

public void setCallbackGuardEnabled(boolean enabled)

Enables or disables try/catch logic for observer callbacks

In order to protect itself from application exceptions in observer callbacks, the Provider normally guards all invocations of application interfaces (e.g. observers) with the following code:

```
try {
    observer.callStateChanged ( ... );
}
```

```catch ( Throwable t ) {
// log the exception here
}
```

This isolates application errors from the JTAPI implementation, allowing easier troubleshooting, since the JTAPI implementation can note the unhandled exception and continue operating. Some errors are considered non-recoverable and will be re-thrown by JTAPI, generally resulting in application exit. Such errors include ThreadDeath, OutOfMemoryError, and StackOverflowError. Applications wishing to trap errors within JTAPI threads should create a subclass of ThreadGroup and initialize JTAPI from a thread within that ThreadGroup. By overriding the ThreadGroup.uncaughtException () method, the application can be made aware of all unrecoverable errors thrown on JTAPI threads.

In some cases, JTAPI’s aggressive error-catching approach may make it more difficult to troubleshoot applications within a java debugger. Microsoft Visual J++ version 6.0, for example, does not handle breakpoints within application observer callbacks properly if JTAPI catches Throwable. In such cases, JTAPI application developers may choose to disable the internal JTAPI try/catch logic.
Disabling callback guards in this manner is only intended for use while troubleshooting applications, and never for use in production environments. By default, callback guards are always enabled.

Parameters:

- enabled: if true, callback guard will be enabled; if false, callback guard will be disabled

unregisterFeature(int)

public void unregisterFeature(int featureID)
throws InvalidStateException

used to unregister a particular feature. Provider events for the feature will stop after unregistering the feature

Throws:

javax.telephony.InvalidStateException

CiscoProviderCapabilities

Declaration

public interface CiscoProviderCapabilities extends javax.telephony.capabilities.ProviderCapabilities

All Superinterfaces

javax.telephony.capabilities.ProviderCapabilities

Description

This interface defines the specific capabilities offered by the Cisco Jtapi implementation.
CiscoProviderObserver

Declaration

public interface CiscoProviderObserver extends javax.telephony.ProviderObserver

All Superinterfaces

javax.telephony.ProviderObserver

Description

Applications implement this interface in order to receive CiscoProvEv events such as CiscoAddrCreatedEv and CiscoTermCreatedEv when observing a Provider via the Provider.addObserver method.

See Also:
CiscoAddrCreatedEv, CiscoTermCreatedEv

Inherited Member Summary

Methods inherited from interface ProviderCapabilities
isObservable()
CiscoRegistrationException

Declaration

```java
public class CiscoRegistrationException extends java.lang.Exception
```

All Implemented Interfaces

java.io.Serializable

Description

The CiscoMediaTerminal.register method throws this exception when the registration process fails for any reason. For example, registration would fail if the Provider were OUT_OF_SERVICE or if the device were already registered.

See Also: CiscoMediaTerminal.register(InetAddress, int, CiscoMediaCapability[])
Member Summary

Constructors

- CiscoRegistrationException()
- CiscoRegistrationException(java.lang.String description)

Inherited Member Summary

Methods inherited from class Object

- clone(), equals(Object), finalize(), getClass(), notify(), notifyAll(),
- wait(), wait(), wait()

Methods inherited from class Throwable

- fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(),
- initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter),
- printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
All Superinterfaces

javax.telephony.Address,
javax.telephony.callcenter.RouteAddress

Member Summary

Methods

void registerRouteCallback(javax.telephony.callcenter.RouteCallback routeCallback, boolean disableAutoRehoming)

Inherited Member Summary

Fields inherited from interface RouteAddress

ALL_ROUTE_ADDRESS

Methods inherited from interface Address

addCallObserver(CallObserver), addObserver(AddressObserver), getAttribute(Terminal), getCallObservers(), getCapabilities(), getConnections(), getName(), getObservers(), getProvider(), getTerminals(), removeCallObserver(CallObserver), removeObserver(AddressObserver)

Methods inherited from interface RouteAddress

cancelRouteCallback(RouteCallback), getActiveRouteSessions(), getRouteCallback(), registerRouteCallback(RouteCallback)

Methods

registerRouteCallback(RouteCallback, boolean)

public void registerRouteCallback(javax.telephony.callcenter.RouteCallback routeCallback, boolean disableAutoRehoming)
Chapter 2  Cisco JTAPI Implementation

CiscoRouteSession

Declaration

public interface CiscoRouteSession extends javax.telephony.callcenter.RouteSession

All Superinterfaces

javax.telephony.callcenter.RouteSession

Description

Introduction
The CiscoRouteSession supports application access to underlying call associated with a RouteSession. Also, various internal ERRORs where endRoute is called internally are exposed to the app, should they wish to handle endRouteEvent() in any special way for these cases.

See Also:
javax.telephony.Call

Member Summary

Fields

boolean disableAutoRehoming)
throws ResourceUnavailableException, MethodNotSupportedException

Throws:
javax.telephony.MethodNotSupportedException,
javax.telephony.ResourceUnavailableException
Chapter 2      Cisco JTAPI Implementation

CiscoRouteSession

Member Summary

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLINGADDRESS_SEARCH_SPACE</td>
<td>This indicates that the redirect should be done using the search space of the calling address.</td>
</tr>
<tr>
<td>DEFAULT_SEARCH_SPACE</td>
<td>This indicates that the redirect should be done using the search space that is the default for the implementation.</td>
</tr>
<tr>
<td>ERROR_INVALID_STATE</td>
<td>If an internal InvalidStateException occurred or some preconditions/postconditions were not met during routing, endRoute is called with this ERROR_INVALID_STATE error.</td>
</tr>
<tr>
<td>ERROR_NO_CALLBACK</td>
<td>For now, since there is no default route mechanism in place, if there is no callback registered for this app, an endRoute with this error is called.</td>
</tr>
<tr>
<td>ERROR_ROUTESELECT_TIMEOUT</td>
<td>Each routeEvent()/reRouteEvent() sent starts a timer for the app to respond with a routeSelect()/endRoute().</td>
</tr>
<tr>
<td>ROUTEADDRESS_SEARCH_SPACE</td>
<td>This indicates that the redirect should be done using the search space of the route point address.</td>
</tr>
</tbody>
</table>

Methods

javax.telephony.Call getCall()
Returns the call associated with this RouteSession.

void selectRoute(java.lang.String routeSelected, int callingSearchSpace)
This method overloads the selectRoute method in the RouteSession interface to allow applications to specify a calling search space to be used when the call is redirected to the route destination.

Inherited Member Summary

Fields inherited from interface RouteSession
### Inherited Member Summary

<table>
<thead>
<tr>
<th>Member Summary</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAUSE_INVALID_DESTINATION, CAUSE_NO_ERROR, CAUSE_PARAMETER_NOT_SUPPORTED,</td>
<td>This indicates that the redirect should be done using the search space of the</td>
</tr>
<tr>
<td>CAUSE_ROUTING_TIMER_EXPIRED, CAUSE_STATE_INCOMPATIBLE, CAUSE_UNSPECIFIED_ERROR,</td>
<td>calling address.</td>
</tr>
<tr>
<td>ERROR_RESOURCE_BUSY, ERROR_RESOURCE_OUT_OF_SERVICE, ERROR_UNKNOWN, RE_ROUTE,</td>
<td></td>
</tr>
<tr>
<td>ROUTE, ROUTE_CALLBACK_ENDED, ROUTE_END, ROUTE_USED</td>
<td></td>
</tr>
</tbody>
</table>

### Fields

**CALLINGADDRESS_SEARCH_SPACE**

```
public static final int CALLINGADDRESS_SEARCH_SPACE
```

This indicates that the redirect should be done using the search space of the calling address.

**DEFAULT_SEARCH_SPACE**

```
public static final int DEFAULT_SEARCH_SPACE
```

This indicates that the redirect should be done using the search space that is the default for the implementation. The default is to use the caller’s search space.

**ERROR_INVALID_STATE**

```
public static final int ERROR_INVALID_STATE
```

If an internal InvalidStateException occurred or some preconditions/postconditions were not met during routing endRoute is called with this ERROR_INVALID_STATE error.

**ERROR_NO_CALLBACK**

```
public static final int ERROR_NO_CALLBACK
```

For now, since there is no default route mechanism in place, if there is no callback register for this app, an endRoute with this error is called.
ERROR_NONE

public static final int ERROR_NONE

ERRORS defined for internal successful endRoute call. Error value set for no error.

ERROR_ROUTESELECT_TIMEOUT

public static final int ERROR_ROUTESELECT_TIMEOUT

Each routeEvent()/reRouteEvent() sent starts a timer for the app to respond with a routeSelect()/endRoute(). The default value of this timer is 5secs. Should the application not respond within this time, an endRoute is called with this error = ERROR_ROUTESELECT_TIMEOUT

ROUTEADDRESS_SEARCH_SPACE

public static final int ROUTEADDRESS_SEARCH_SPACE

This indicates that the redirect should be done using the search space of the route point address.

Methods

getCall()

public javax.telephony.Call getCall()

Returns the call associated with this RouteSession.

Returns: the call associated with this RouteSession

selectRoute(String[], int)

public void selectRoute(java.lang.String[] routeSelected, int callingSearchSpace)

throws MethodNotSupportedException

This method overloads the selectRoute method in the RouteSession interface to allow applications to specify a calling search space to be used when the call is redirected to the route destination.
The callingSearchSpace parameter may be:

1. CiscoRouteSession.DEFAULT_SEARCH_SPACE
2. CiscoRouteSession.CALLINGADDRESS_SEARCH_SPACE
3. CiscoRouteSession.ROUTEADDRESS_SEARCH_SPACE

Read above for a description of what each of these means.

Throws:
javax.telephony.MethodNotSupportedException

CiscoRTPBitRate

Declaration

public interface CiscoRTPBitRate

Description

The RTPBitRate interface contains constants describing G.723 RTP bitrates. These constants are returned by the CiscoRTPInputProperties.getBitRate method and the CiscoRTPOutputProperties.getBitRate method.

See Also:
CiscoRTPInputProperties.getBitRate(), CiscoRTPOutputProperties.getBitRate()

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int R5_3</td>
</tr>
<tr>
<td>static int R6_4</td>
</tr>
</tbody>
</table>
### Fields

**R5_3**

```
public static final int R5_3
5.3k G.723 bit rate
```

**R6_4**

```
public static final int R6_4
6.4k G.723 bit rate
```

### CiscoRTPInputProperties

**Declaration**

```
public interface CiscoRTPInputProperties
```

### Description

**Member Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getBitRate()</code></td>
<td>Returns the media bit rate, one of the following constants:</td>
</tr>
<tr>
<td><code>boolean getEchoCancellation()</code></td>
<td></td>
</tr>
<tr>
<td><code>java.net.InetAddress getLocalAddress()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getLocalPort()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getPacketSize()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getPayloadType()</code></td>
<td>Returns the payload format, one of the following constants:</td>
</tr>
</tbody>
</table>
Methods

getBitRate()
public int getBitRate()

Returns the media bit rate, one of the following constants:

- CiscoRTPBitRate.R5_3
- CiscoRTPBitRate.R6_4

Returns: payload type

gEchoCancellation()
public boolean getEchoCancellation()

Returns: echo cancellation

gLocalAddress()
public java.net.InetAddress getLocalAddress()

Returns: address to which media will be directed

gLocalPort()
public int getLocalPort()

Returns: port to which media will be directed

gPacketSize()
public int getPacketSize()

Returns: packet size, in milliseconds

gPayloadType()
public int getPayloadType()

Returns the payload format, one of the following constants:

- CiscoRTPPayload.NONSTANDARD
- CiscoRTPPayload.G711ALAW64K
- CiscoRTPPayload.G711ALAW56K
- CiscoRTPPayload.G711ULAW64K
CiscoRTPInputStartedEv

Declaration

public interface CiscoRTPInputStartedEv extends CiscoTermEv

All Superinterfaces

CiscoEv, CiscoTermEv, javax.telephony.events.Ev, javax.telephony.events.TermEv

Returns: payload type
CiscoRTPInputStartedEv

## Description

### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td>CiscoCallID getCallID() returns CiscoCallID</td>
</tr>
<tr>
<td></td>
<td>CiscoRTPInputProperties getRTPInputProperties()</td>
</tr>
</tbody>
</table>

### Inherited Member Summary

#### Fields inherited from interface Ev

- CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

#### Methods inherited from interface Ev

- getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

#### Methods inherited from interface TermEv

- getTerminal()
Fields

ID

    public static final int ID

Methods

getCallID()

    public com.cisco.jtapi.extensions.CiscoCallID getCallID()
    returns CiscoCallID

getRTPInputProperties()

    public com.cisco.jtapi.extensions.CiscoRTPInputProperties
        getRTPInputProperties()

    Returns: RTP input properties

CiscoRTPInputStoppedEv

Declaration

    public interface CiscoRTPInputStoppedEv extends CiscoTermEv

All Superinterfaces

    CiscoEv, CiscoTermEv, javax.telephony.events.Ev,
    javax.telephony.events.TermEv

Description
Member Summary

Fields

static int ID

Methods

CiscoCallID getCallID()
    returns CiscoCallID

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getausCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv

getTerminal()

Fields

ID

public static final int ID
Chapter 2      Cisco JTAPI Implementation

CiscoRTPOutputProperties

Methods

getCodeID()

    public com.cisco.jtapi.extensions.CiscoCallID getCodeID()
    returns CiscoCallID

CiscoRTPOutputProperties

Declaration

    public interface CiscoRTPOutputProperties

Description

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getBitRate()</td>
</tr>
<tr>
<td>int getMaxFramesPerPacket()</td>
</tr>
<tr>
<td>int getPacketSize()</td>
</tr>
<tr>
<td>int getPayloadType()</td>
</tr>
<tr>
<td>java.net.InetAddress getRemoteAddress()</td>
</tr>
<tr>
<td>int getRemotePort()</td>
</tr>
<tr>
<td>boolean getSilenceSuppression()</td>
</tr>
</tbody>
</table>

Returns the media bit rate, one of the following constants:

Returns the payload format, one of the following constants:
Methods

getBitRate()

```java
public int getBitRate()
```

Returns the media bit rate, one of the following constants:

- `CiscoRTPBitRate.R5_3`
- `CiscoRTPBitRate.R6_4`

Returns: payload type

getMaxFramesPerPacket()

```java
public int getMaxFramesPerPacket()
```

Returns: the maximum number of frames to send per packet

getPacketSize()

```java
public int getPacketSize()
```

Returns: packet size, in milliseconds

getPayloadType()

```java
public int getPayloadType()
```

Returns the payload format, one of the following constants:

- `CiscoRTPPayload.NONSTANDARD`
- `CiscoRTPPayload.G711ALAW64K`
- `CiscoRTPPayload.G711ALAW56K`
- `CiscoRTPPayload.G711ULAW64K`
- `CiscoRTPPayload.G711ULAW56K`
- `CiscoRTPPayload.G722_64K`
- `CiscoRTPPayload.G722_56K`
- `CiscoRTPPayload.G722_48K`
- `CiscoRTPPayload.G7231`
- `CiscoRTPPayload.G728`
CiscoRTPOutputProperties

- CiscoRTPPayload.G729
- CiscoRTPPayload.G729ANNEXA
- CiscoRTPPayload.IS11172AUDIOCAP
- CiscoRTPPayload.IS13818AUDIOCAP
- CiscoRTPPayload.ACY_G729AASSN
- CiscoRTPPayload.DATA64
- CiscoRTPPayload.DATA56
- CiscoRTPPayload.GSM
- CiscoRTPPayload.ACTIVEVOICE

Returns: payload type

getPrecedenceValue()

public int getPrecedenceValue()

Returns: precedence value

getRemoteAddress()

public java.net.InetAddress getRemoteAddress()

Returns: address to which media is to be transmitted

getRemotePort()

public int getRemotePort()

Returns: port to which media is to be transmitted

getSilenceSuppression()

public boolean getSilenceSuppression()

Returns: silence suppression
CiscoRTPOutputStartedEv

Declaration

```java
public interface CiscoRTPOutputStartedEv extends CiscoTermEv
```

All Superinterfaces

- CiscoEv, CiscoTermEv, javax.telephony.events.Ev, javax.telephony.events.TermEv

Description

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td>static int ID</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>CiscoCallID getCallID() returns CiscoCallID</td>
</tr>
<tr>
<td>CiscoRTPOutputProperties getRTPOutputProperties()</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Fields inherited from interface Ev
Inherited Member Summary

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv

geterminal()

Fields

public static final int ID

Methods

public com.cisco.jtapi.extensions.CiscoCallID getCallID()

returns CiscoCallID

public com.cisco.jtapi.extensions.CiscoRTPOutputProperties
getRTPOutputProperties()

Returns: RTP output properties
CiscoRTPOutputStoppedEv

Declaration

public interface CiscoRTPOutputStoppedEv extends CiscoTermEv

All Superinterfaces

CiscoEv, CiscoTermEv, javax.telephony.events.Ev, javax.telephony.events.TermEv

Description

Member Summary

Fields

static int ID

Methods

CiscoCallID getCallID()
returns CiscoCallID

Inherited Member Summary

Fields inherited from interface Ev
CiscoRTPParams

Inherited Member Summary

- CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
  CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
  CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
  CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
  META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
  META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev
- getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv
- getTerminal()

Fields

- public static final int ID

Methods

- public com.cisco.jtapi.extensions.CiscoCallID getCallID()
  returns CiscoCallID

CiscoRTPParams

Declaration

- public class CiscoRTPParams
CiscoRTPParams

java.lang.Object
|-- com.cisco.jtapi.extensions.CiscoRTPParams

### Member Summary

#### Constructors

- `CiscoRTPParams(java.net.InetAddress rtpAddress, int rtpPort)

#### Methods

- `java.net.InetAddress getRTPAddress()
- `java.lang.String getRTPAddressHostName()
- `byte[] getRTPByteAddress()
- `int getRTPPort()
- `java.lang.String toString()

### Inherited Member Summary

Methods inherited from class `Object`

- `clone()`, `equals(Object)`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `wait()`, `wait()`, `wait()`

### Constructors

```
public CiscoRTPParams(java.net.InetAddress rtpAddress, int rtpPort)
```

### Methods

```
public java.net.InetAddress getRTPAddress()
```
getRTPAddressHostName()
   public java.lang.String getRTPAddressHostName()

getRTPByteAddress()
   public byte[] getRTPByteAddress()

getRTPPort()
   public int getRTPPort()

toString()
   public java.lang.String toString()
   Overrides: toString in class Object

CiscoRTPPayload

Declaration

public interface CiscoRTPPayload

Description

The RTPPayload interface contains constants describing RTP formats. These constants are returned by the CiscoRTPInputProperties.getPayloadType method and the CiscoRTPOutputProperties.getPayloadType method.

See Also:
   CiscoRTPInputProperties.getPayloadType(), CiscoRTPOutputProperties.getPayloadType()
### Member Summary

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVEVOICE</td>
<td>ACTIVEVOICE payload</td>
</tr>
<tr>
<td>ACY_G729AASSN</td>
<td>ACY_G729AASSN payload</td>
</tr>
<tr>
<td>DATA56</td>
<td>DATA56 payload</td>
</tr>
<tr>
<td>DATA64</td>
<td>DATA64 payload</td>
</tr>
<tr>
<td>G711ALAW56K</td>
<td>G.711 56K a-law payload</td>
</tr>
<tr>
<td>G711ALAW64K</td>
<td>G.711 64K a-law payload</td>
</tr>
<tr>
<td>G711ULAW56K</td>
<td>G.711 56K u-law payload</td>
</tr>
<tr>
<td>G711ULAW64K</td>
<td>G.711 64K u-law payload</td>
</tr>
<tr>
<td>G722_48K</td>
<td>G.722 48K payload</td>
</tr>
<tr>
<td>G722_56K</td>
<td>G.722 56K payload</td>
</tr>
<tr>
<td>G722_64K</td>
<td>G.722 64K payload</td>
</tr>
<tr>
<td>G7231</td>
<td>G.723.1 payload</td>
</tr>
<tr>
<td>G728</td>
<td>G.728 payload</td>
</tr>
<tr>
<td>G729</td>
<td>G.729 payload</td>
</tr>
<tr>
<td>G729ANEXA</td>
<td>G.729a payload</td>
</tr>
<tr>
<td>GSM</td>
<td>GSM payload</td>
</tr>
<tr>
<td>IS11172AUDIOCAP</td>
<td>IS11172AUDIOCAP payload</td>
</tr>
<tr>
<td>IS13818AUDIOCAP</td>
<td>IS13818AUDIOCAP payload</td>
</tr>
<tr>
<td>NONSTANDARD</td>
<td>A non-standard RTP payload</td>
</tr>
</tbody>
</table>
Fields

**ACTIVEVOICE**

    public static final int ACTIVEVOICE

    ACTIVEVOICE payload

**ACY_G729AASSN**

    public static final int ACY_G729AASSN

    ACY_G729AASSN payload

**DATA56**

    public static final int DATA56

    DATA56 payload

**DATA64**

    public static final int DATA64

    DATA64 payload

**G711ALAW 56K**

    public static final int G711ALAW56K

    G.711 56K a-law payload

**G711ALAW 64K**

    public static final int G711ALAW64K

    G.711 64K a-law payload

**G711ULAW 56K**

    public static final int G711ULAW56K

    G.711 56K u-law payload

**G711ULAW 64K**

    public static final int G711ULAW64K

    G.711 64K u-law payload
public static final int G722_48K
G.722 48K payload

public static final int G722_56K
G.722 56K payload

public static final int G722_64K
G.722 64K payload

public static final int G7231
G.723.1 payload

public static final int G728
G.728 payload

public static final int G729
G.729 payload

public static final int G729ANNEXA
G.729a payload

public static final int GSM
GSM payload

public static final int IS11172AUDIOCAP
IS11172AUDIOCAP
Cisco JTAPI Implementation

Chapter 2

CiscoSynchronousObserver

Declaration

public interface CiscoSynchronousObserver

Description

The Cisco JT API implementation is designed to allow applications to invoke blocking JT API methods such as Call.connect() and TerminalConnection.answer() from within their observer callbacks. This means that applications are not subject to the restrictions imposed by the JT API specification, which cautions applications against using JT API methods from within observer callbacks.

Normally, when an application adds a new observer to a JTAPI object, the Cisco JT API implementation creates an event queue and an accompanying worker thread to service the new observer. If the same observer is added to another object, its queue and thread are reused; in effect, every unique observer object has a single queue and worker thread. As noted, the advantage of this arrangement is that an application may invoke blocking JTAPI methods from within its observer callback. A subtle disadvantage, however, is that accessor methods such as Call.getConnections() and

IS11172AUDIOCAP payload

public static final int IS11172AUDIOCAP

IS13818AUDIOCAP payload

public static final int IS13818AUDIOCAP

NONSTANDARD

public static final int NONSTANDARD

A non-standard RTP payload
Connection.getState() may not return results that are consistent with events when invoked from within the observer callback.

For example, suppose that an application creates and connects a call from address “A” to address “B”. If the application is observing address “A”, it might reasonably expect that when it receives the CallActiveEv, the state of the call will be Call.ACTIVE. This is not necessarily so, since the worker thread delivering events to the application is decoupled from the internal JTAPI thread that is updating object states. In fact, if “B” rejects the call from “A”, the call object might be in either the Call.ACTIVE state or the Call.INVALID state, depending on the exact moment at which the worker thread delivers the CallActiveEv.

Many applications will not be adversely affected by this asynchronous behavior. Applications that would benefit from a coherent call model during observer callbacks, however, can selectively disable the queueing logic of the Cisco JTAPI implementation. By implementing the CiscoSynchronousObserver interface on its observer objects, an application declares that it wishes events to be delivered synchronously to its observers. Events delivered to synchronous observers will match the states of the call model objects queried from within the observer callback.

**Usage Notes**

Objects that implement the CiscoSynchronousObserver interface are strictly forbidden from invoking blocking JTAPI methods from within their event callbacks. The consequences of doing so are unpredictable, and may include deadlocking the JTAPI implementation. On the other hand, they may safely use the accessor methods of any JTAPI object, for instance Call.getConnections() or Connection.getState().

---

**CiscoTermCreatedEv**

**Declaration**

```java
public interface CiscoTermCreatedEv extends CiscoProvEv
```

---
Chapter 2      Cisco JTAPI Implementation

All Superinterfaces

CiscoEv, CiscoProvEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv

Description

The CiscoTermCreatedEv event

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTerminal()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inherited Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields inherited from interface Ev</td>
</tr>
<tr>
<td>CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from interface Ev</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from interface ProvEv</th>
</tr>
</thead>
</table>

Cisco J TAPI Developer Guide for Cisco CallManager 3.3(2)
CiscoTermDataEv

Inherited Member Summary

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getProvider()</td>
</tr>
</tbody>
</table>

Fields

ID

public static final int ID

Methods

getTerminal()

public javax.telephony.Terminal getTerminal()

CiscoTermDataEv

Declaration

public interface CiscoTermDataEv extends CiscoTermEv

All Superinterfaces

CiscoEv, CiscoTermEv, javax.telephony.events.Ev, javax.telephony.events.TermEv

Description

Introduction

The CiscoTermDataEv event
Member Summary

Fields

static int ID

Methods

java.lang.String getData()

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv

getTerminal()

Fields

ID

public static final int ID
CiscoTermEv

Methods

dataGet()

public java.lang.String getData()

CiscoTermEv

Declaration

public interface CiscoTermEv extends CiscoEv,
javax.telephony.events.TermEv

All Superinterfaces

CiscoEv, javax.telephony.events.Ev,
javax.telephony.events.TermEv

All Known Subinterfaces

CiscoRTPInputStartedEv, CiscoRTPInputStoppedEv,
CiscoRTPOutputStartedEv, CiscoRTPOutputStoppedEv,
CiscoTermDataEv, CiscoTermInServiceEv,
CiscoTermOutOfServiceEv

Description

The CiscoTermEv interface, which extends JTAPI’s core
javax.telephony.events.TermEv interface, serves as the base
interface for all Cisco-extended JTAPI Terminal events. Every Call-related event
in this package extends this interface, directly or indirectly.
Chapter 2  Cisco JTAPI Implementation

See Also:
javax.telephony.events.CallEv

Inherited Member Summary

Fields inherited from interface Ev
CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOTAVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALLENDING, META_CALLMERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev
getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv
getTerminal()

CiscoTerminal

Declaration

public interface CiscoTerminal extends javax.telephony.Terminal,
CiscoObjectContainer

All Superinterfaces

CiscoObjectContainer,javax.telephony.Terminal
All Known Subinterfaces

CiscoMediaTerminal

Description

Since JTAPI does not support the notion of dynamic terminal registration, the CiscoTerminal interface extends the standard Terminal interface to do so. All CallManager devices are represented by CiscoTerminals, and all CiscoTerminals may be queried to determine whether they are currently IN_SERVICE or OUT_OF_SERVICE.

If the CallManager device represented by the CiscoTerminal is an IP telephone, for instance, it would become OUT_OF_SERVICE if it were to lose its network connection. Other types of devices, such as CTIPorts, are registered on demand by applications, and may be IN_SERVICE or OUT_OF_SERVICE accordingly.

See Also:
javax.telephony.Terminal, CiscoMediaTerminal

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int IN_SERVICE</td>
</tr>
<tr>
<td>static int OUT_OF_SERVICE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getRegistrationState()</td>
</tr>
<tr>
<td>Returns the state of this terminal.</td>
</tr>
<tr>
<td>CiscoRTPInputProperties()</td>
</tr>
<tr>
<td>The CiscoTerminal must be in the CiscoTerminal.REGISTERED state, its Provider must be in the Provider.IN_SERVICE state.</td>
</tr>
<tr>
<td>CiscoRTPOutputProperties()</td>
</tr>
<tr>
<td>The CiscoTerminal must be in the CiscoTerminal.REGISTERED state, its Provider must be in the Provider.IN_SERVICE state.</td>
</tr>
<tr>
<td>int getState()</td>
</tr>
<tr>
<td>Returns the state of this terminal.</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

CiscoTerminal

Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String sendData(java.lang.String terminalData)</td>
<td>This method sends data to the terminal.</td>
</tr>
<tr>
<td>javax.telephony.TerminalConnection unPark(javax.telephony.Address UnParkAddress, java.lang.String ParkedAt)</td>
<td>This method unparks a call at a terminal and returns a terminalConnection.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

- Methods inherited from interface CiscoObjectContainer: `getObject()`, `setObject(Object)`
- Methods inherited from interface Terminal: `addCallObserver(CallObserver)`, `addObserver(TerminalObserver)`, `getAddresses()`, `getCallObservers()`, `getCapabilities()`, `getName()`, `getObservers()`, `getProvider()`, `getTerminalCapabilities(Terminal, Address)`, `getTerminalConnections()`, `removeCallObserver(CallObserver)`, `removeObserver(TerminalObserver)`

Fields

- `IN_SERVICE`: public static final int
- `OUT_OF_SERVICE`: public static final int

Methods

- `getRegistrationState()`: public int
  - **Deprecated.** This method has been replaced by the `getState()` method.
  - Returns the state of this terminal.
The state may be any of the following constants:

- CiscoTerminal.OUT_OF_SERVICE
- CiscoTerminal.IN_SERVICE

**Returns:** the state of this terminal

```java
getRTPInputProperties()
public com.cisco.jtapi.extensions.CiscoRTPInputProperties
    getRTPInputProperties()
    throws InvalidStateException
```

The CiscoTerminal must be in the CiscoTerminal.REGISTERED state, its Provider must be in the Provider.IN_SERVICE state, and Terminal.getTerminalConnections () must return at least one terminal connection in the TerminalConnection.ACTIVE state.

**Returns:** the properties to be used for the RTP input stream associated with the ACTIVE TerminalConnection on this Terminal

**Throws:**

```
javax.telephony.InvalidStateException
```

```java
getRTPOutputProperties()
public com.cisco.jtapi.extensions.CiscoRTPOutputProperties
    getRTPOutputProperties()
    throws InvalidStateException
```

The CiscoTerminal must be in the CiscoTerminal.REGISTERED state, its Provider must be in the Provider.IN_SERVICE state, and Terminal.getTerminalConnections () must return at least one terminal connection in the TerminalConnection.ACTIVE state.

**Returns:** the properties to be used for the RTP output stream associated with the ACTIVE TerminalConnection on this Terminal

**Throws:**

```
javax.telephony.InvalidStateException
```

```java
getState()
public int getState()
```

Returns the state of this terminal.
The state may be any of the following constants:

- `CiscoTerminal.OUT_OF_SERVICE`
- `CiscoTerminal.IN_SERVICE`

**Returns:** the state of this terminal

**sendData(String)**

```java
public java.lang.String sendData(java.lang.String terminalData)
throws InvalidStateException, MethodNotSupportedException
```

**Throws:**

- `javax.telephony.MethodNotSupportedException`
- `javax.telephony.InvalidStateException`

**unPark(Address, String)**

```java
public javax.telephony.TerminalConnection unPark(javax.telephony.Address UnParkAddress,
java.lang.String ParkedAt)
throws InvalidStateException, IllegalArgumentException,
ResourceUnavailableException
```

This method Unparks a call at a terminal and returns a terminalConnection. The `CiscoTerminal` must be in `CiscoTerminal.REGISTERED` state, its Provider must be in the `Provider.IN_SERVICE` state. This method takes an address and string as input. The address is any address on the terminal and string is system park port where a call was previously parked. This string is returned when the a call is parked

**Throws:**

- `javax.telephony.ResourceUnavailableException`
- `javax.telephony.InvalidArgument Exception`
- `javax.telephony.InvalidStateException`
CiscoTerminalConnection

Declaration

```java
public interface CiscoTerminalConnection extends javax.telephony.callcontrol.CallControlTerminalConnection, CiscoObjectContainer
```

All Superinterfaces

javax.telephony.callcontrol.CallControlTerminalConnection, CiscoObjectContainer, javax.telephony.TerminalConnection

Description

The CiscoTerminalConnection interface extends the CallControlTerminalConnection interface with additional capabilities.

Applications can use the `getReason` method to obtain the reason for the creation of this Connection.

### Inherited Member Summary

<table>
<thead>
<tr>
<th>Fields inherited from interface CallControlTerminalConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRIDGED, DROPPED, HELD, IDLE, INUSE, RINGING, TALKING, UNKNOWN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fields inherited from interface TerminalConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE, PASSIVE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from interface CallControlTerminalConnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCallControlState(), hold(), join(), leave(), unhold()</td>
</tr>
</tbody>
</table>
CiscoTerminalObserver

Declaration

public interface CiscoTerminalObserver extends javax.telephony.TerminalObserver

All Superinterfaces:

javax.telephony.TerminalObserver

Description

Applications implement this interface in order to receive CiscoTermEv events such as CiscoRTPInputStartedEv and CiscoRTPInputStoppedEv when observing Terminals via the Terminal.addObserver method.

See Also:

CiscoTermInServiceEv, CiscoTermOutOfServiceEv,
CiscoRTPInputStartedEv, CiscoRTPInputStoppedEv,
CiscoRTPOutputStartedEv, CiscoRTPOutputStoppedEv

Inherited Member Summary

Methods inherited from interface CiscoObjectContainer

getObject(), setObject(Object)

Methods inherited from interface TerminalConnection

answer(), getCapabilities(), getConnection(), getState(), getTerminal(),
getTerminalConnectionCapabilities(Terminal, Address)
## CiscoTermInServiceEv

**Declaration**

```java
description
public interface CiscoTermInServiceEv extends CiscoTermEv
```

**All Superinterfaces**

- `CiscoEv`, `CiscoTermEv`, `javax.telephony.events.Ev`, `javax.telephony.events.TermEv`

**Description**

### Introduction

The `CiscoTermInServiceEv` event

**Member Summary**

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
</tr>
</tbody>
</table>
Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface TermEv

getTerminal()

Fields

ID

public static final int ID

CiscoTermOutOfServiceEv

Declaration

public interface CiscoTermOutOfServiceEv extends CiscoTermEv, CiscoOutOfServiceEv
All Superinterfaces

CiscoEv, CiscoOutOfServiceEv, CiscoTermEv, javax.telephony.events.Ev, javax.telephony.events.TermEv

Description

The CiscoTermOutOfServiceEv event

Member Summary

Fields

| static int ID |

Inherited Member Summary

Fields inherited from interface CiscoOutOfServiceEv

| CAUSE_CALLMANAGER_FAILURE, CAUSE_CTIMANAGER_FAILURE, CAUSE_DEVICE_FAILURE, CAUSE_DEVICE_UNREGISTERED, CAUSE_NOCALLMANAGER_AVAILABLE, CAUSE_REHOMING_TO_HIGHER_PRIORITY_CM, CAUSE_REHOMING_FAILURE |

Fields inherited from interface Ev

| CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN |

Methods inherited from interface Ev

| getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent() |
Chapter 2      Cisco J TAPI Implementation

CiscoTermRemovedEv

Inherited Member Summary

Methods inherited from interface TermEv
getTerminal()

Fields

ID
public static final int ID

CiscoTermRemovedEv

Declaration

public interface CiscoTermRemovedEv extends CiscoProvEv

All Superinterfaces

CiscoEv, CiscoProvEv, javax.telephony.events.Ev, javax.telephony.events.ProvEv

Description

Introduction
The CiscoTermRemovedEv event

Member Summary

Fields
Member Summary

| static int | ID |

Methods

getTerminal()

javax.telephony.Terminal

Inherited Member Summary

Fields inherited from interface Ev

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION, CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE, CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT, CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING, META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING, META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface Ev

getausCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Methods inherited from interface ProvEv

getProvider()

Fields

ID

public static final int ID
# Methods

getTerminal()

```java
generic javax.telephony.Terminal getTerminal()
```

## CiscoTransferEndEv

### Declaration

```java
public interface CiscoTransferEndEv extends CiscoCallEv
```

### All Superinterfaces

- `javax.telephony.events.CallEv`
- `CiscoCallEv`
- `CiscoEv`
- `javax.telephony.events.Ev`

## Description

The `CiscoTransferEndEv` event indicates that a transfer operation has completed. This event is reported via the `CallControlCallObserver` interface.

## Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
<td>javax.telephony.Call getFinalCall()</td>
</tr>
</tbody>
</table>

Returns the call that remains active after the transfer is completed.
## Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>javax.telephony.TerminalConnection</strong></td>
<td>getTransferController()</td>
</tr>
<tr>
<td>Returns the TerminalConnection which currently acts as the transfer controller for this call — the final call.</td>
<td></td>
</tr>
<tr>
<td><strong>javax.telephony.Address</strong></td>
<td>getTransferControllerAddress()</td>
</tr>
<tr>
<td>Returns the Address which currently acts as the transfer controller for this call — the final call.</td>
<td></td>
</tr>
<tr>
<td><strong>javax.telephony.Call</strong></td>
<td>getTransferredCall()</td>
</tr>
<tr>
<td>Returns the call that has been transferred.</td>
<td></td>
</tr>
</tbody>
</table>

## Inherited Member Summary

### Fields inherited from interface `CiscoCallEv`

- CAUSE_ACCESSINFORMATIONDISCARDED
- CAUSE_BCBPRESENTLYAVAIL
- CAUSE_BCBNOTAUTHORIZED
- CAUSE_BEARERCAPNIMPL
- CAUSE_CALLBEINGDELIVERED
- CAUSE_CALLIDINUSE
- CAUSE_CALLMANAGER_FAILURE
- CAUSE_CALLREJECTED
- CAUSE_CALLSPLIT
- CAUSE_CHANUNACCEPTABLE
- CAUSE_CHANMANAGER_FAILURE
- CAUSE_DESTINATIONOUTOFORDER
- CAUSE_DESTNUMMISSANDDCNOTSUB
- CAUSE_FACILITYREJECTED
- CAUSE_IDENTIFIEDCHANDOESNOTEXIST
- CAUSE_IENIMPL
- CAUSE_INBOUNDBLINDTRANSFER
- CAUSE_INBOUNDCONFERENCE
- CAUSE_INBOUNDTRANSFER
- CAUSE_INCOMINGCALLBARRED
- CAUSE_INCOMPATABLEDESTINATION
- CAUSE_INTERWORKINGUNSPECIFIED
- CAUSE_INVALIDCALLREFVALUE
- CAUSE_INVALIDMESSAGEUNSPECIFIED
- CAUSE_INVALIDNUMBERFORMAT
- CAUSE_INVALIDTRANSITNETSEL
- CAUSE_MANDATORYIEMISSING
- CAUSE_MSGNCOMPATABLEWCS
- CAUSE_MSGTYPENCOMPATWCS
- CAUSE_MSGTYPENIMPL
- CAUSE_NETOUTOFORDER
- CAUSE_NOANSWERFROMUSER
- CAUSE_NOCALLSUSPENDED
- CAUSE_NOANSWERFROMUSERCLEARING
- CAUSE_NORMALCALLCLEARING
- CAUSE_NORMALUNSPECIFIED
- CAUSE_NOROUTETOTODESTINATION
- CAUSE_NOROUTETOTRANSITNET
- CAUSE_NouserRESPONDING
- CAUSE_NUMBERCHANGED
- CAUSE_ONLYRDIVEARERCAPNAVAIL
- CAUSE_OUTBOUNDCONFERENCE
- CAUSE_OUTBOUNDTRANSFER
- CAUSE_PROTOCOLERRORUNSPECIFIED
- CAUSE_QALOSERNAVaIL
- CAUSE_RECOVERYONTIMEREXPIRY
- CAUSE_REDIRECTED
- CAUSE_REQCALLIDHASBEENCLEARED
- CAUSE_REQCIRCNAVAIL
- CAUSE_REQFACILITYNIMPL
- CAUSE_REQFACILITYNOTSUBSCRIBED
- CAUSE_RESERVEFAVAILABLE
- CAUSE_RESPORTSTATUSUNQUERY
- CAUSE_SERVNOTAVAILUNSPECIFIED
- CAUSE_SERVOPERATIONVIOlATED
- CAUSE_SERVOROPTNVAILABLE
- CAUSE_SUSPENDEDBATNOTTHISONE
- CAUSE_SWITCHINGEQUIPMENTCONGESTION
- CAUSE_TEMPORARYFAILURE
- CAUSE_UNALLOCATEDNUMBER
- CAUSE_USERBUSY

Fields inherited from interface `Ev`
Inherited Member Summary

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface CallEv

call

Methods inherited from interface CiscoCallEv

ciscoCause

Methods inherited from interface Ev

cause, id, metaCode, observed, isNewMetaEvent

Fields

ID

public static final int ID

Methods

getFinalCall()

public javax.telephony.Call getFinalCall()

Returns the call that remains active after the transfer is completed.

getTransferController()

public javax.telephony.TerminalConnection getTransferController()
Returns the TerminalConnection which currently acts as the transfer controller for this call — the final call. This method returns null if the transfer controller is not being observed.

\[\text{getTransferControllerAddress()}\]

- Returns the Address which currently acts as the transfer controller for this call — the final call.

\[\text{getTransferredCall()}\]

- Returns the call that has been transferred. This call is in the Call.INVALID state.

\textbf{CiscoTransferStartEv}

\textbf{Declaration}

\texttt{public interface CiscoTransferStartEv extends CiscoCallEv}

\textbf{All Superinterfaces}

\texttt{javax.telephony.events.CallEv, CiscoCallEv, CiscoEv, javax.telephony.events.Ev}

\textbf{Description}

The CiscoTransferStartEv event indicates that a transfer operation has started. This event is reported via the CallControlCallObserver interface.
### Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.telephony.Call getFinalCall()</td>
</tr>
<tr>
<td>Returns the call that will remain active after the transfer is completed.</td>
</tr>
</tbody>
</table>

| javax.telephony.TerminalConnection getTransferController() |
| Returns the TerminalConnection which currently acts as the transfer controller for this call — the final call. |

| javax.telephony.Address getTransferControllerAddress() |
| Returns the Address which currently acts as the transfer controller for this call — the final call. |

| javax.telephony.Call getTransferredCall() |
| Returns the call that will be transferred. |

### Inherited Member Summary

Fields inherited from interface CiscoCallEv
Inherited Member Summary

CAUSE_ACCESSINFORMATIONDISCARDED, CAUSE_BCBPRESENTLYAVAIL, CAUSE_BCAUTHORIZED,
CAUSE_BEARERCAPNIMPL, CAUSE_CALLBEINGDELETED, CAUSE_CALLIDINUSE,
CAUSE_CALLMANAGER_FAILURE, CAUSE_CALLREJECTED, CAUSE_CALLSPLIT, CAUSE_CHANUNACCEPTABLE,
CAUSE_CHANUNACCEPTABLE, CAUSE_CHANMANAGER_FAILURE, CAUSE_DESTINATIONOUTOFORDER,
CAUSE_DESTNUMMISSANDDCNOTSUB, CAUSE_FACILITYREJECTED,
CAUSE_IDENTIFIEDCHANDOESNOTEXIST, CAUSE_IENIMPL, CAUSE_INBOUNDBLINDTRANSFER,
CAUSE_INBOUNDCONFERENCE, CAUSE_INBOUNDTRANSFER, CAUSE_INCOMINGCALLBARRED,
CAUSE_INCOMPATABLEDEDESTINATION, CAUSE_INTERWORKINGUNSPECIFIED,
CAUSE_INVALIDCALLREFVALUE, CAUSE_INVALIDDIECONTENTS, CAUSE_INVALIDMESSAGEUNSPECIFIED,
CAUSE_INVALIDNUMBERFORMAT, CAUSE_INVALIDTRANSITNETS, CAUSE_MANDATORYIEMISSING,
CAUSE_MSGNCOMPATABLEWCS, CAUSE_MSGTYPENCOMPATWCS, CAUSE_MSGTYPENIMPL,
CAUSE_NETOUTOFORDER, CAUSE_NOANSWERFROMUSER, CAUSE_NOCALLSNOTIFIED,
CAUSE_NOERROR, CAUSE_NONSELECTEDUSERCLEARING,
CAUSE_NORMALCALLCLEARING, CAUSE_NORMALUNSPECIFIED, CAUSE_NOROUTETODESTINATION,
CAUSE_NOROUTETOTRANSITNET, CAUSE_NOSERVICEUSED, CAUSE_NUMBERCHANGED,
CAUSE_ONLYRDIVEARERCAPAVAIL, CAUSE_OUTBOUNDCONFERENCE, CAUSE_OUTBOUNDTRANSFER,
CAUSE_PROTOCOLERRORUNSPECIFIED, CAUSEQualifierAVAIL, CAUSE_RECOVORNOTEXPIRED,
CAUSE_RECD, CAUSE_REQCALLIDHASBECOMEM, CAUSE_REQCIRCNAVIL,
CAUSE_REQFACILITYINMPL, CAUSE_REQFACILITYNOTSUBSCRIBED, CAUSE_RESOURCESNAVIL,
CAUSE_RESPONSETOSTATUSENQUIRY, CAUSE_SRVNOTAVAILUNSPECIFIED,
CAUSE_SRVOPERATIONVIOLATED, CAUSE_SRVOROPTNOTNAVAILORIPL,
CAUSE_SUSPCALLBUTNOTTHISONE, CAUSE_SWITCHINGEQUIPMENTCONGESTION,
CAUSE_TEMPORARYFAILURE, CAUSE_UNALLOCATEDNUMBER, CAUSE_USERBUSY

Fields inherited from interface \texttt{Ev}

CAUSE_CALL_CANCELLED, CAUSE_DEST_NOT_OBTAINABLE, CAUSE_INCOMPATIBLE_DESTINATION,
CAUSE_LOCKOUT, CAUSE_NETWORK_CONGESTION, CAUSE_NETWORK_NOT_OBTAINABLE,
CAUSE_NEW_CALL, CAUSE_NORMAL, CAUSE_RESOURCES_NOT_AVAILABLE, CAUSE_SNAPSHOT,
CAUSE_UNKNOWN, META_CALL_ADDITIONAL_PARTY, META_CALL_ENDING, META_CALL_MERGING,
META_CALL_PROGRESS, META_CALL_REMOVING_PARTY, META_CALL_STARTING,
META_CALL_TRANSFERRING, META_SNAPSHOT, META_UNKNOWN

Methods inherited from interface \texttt{CallEv}

getCall()

Methods inherited from interface \texttt{CiscoCallEv}

getCiscoCause()

Methods inherited from interface \texttt{Ev}
Chapter 2      Cisco JTAPI Implementation

Inherited Member Summary
getCause(), getID(), getMetaCode(), getObserved(), isNewMetaEvent()

Fields

ID
    public static final int ID

Methods

getFinalCall()
    public javax.telephony.Call getFinalCall()
    Returns the call that will remain active after the transfer is completed.

getTransferController()
    public javax.telephony.TerminalConnection getTransferController()
    Returns the TerminalConnection which currently acts as the transfer controller for this call — the final call. This method returns null if the transfer controller is not being observed.

getTransferControllerAddress()
    public javax.telephony.Address getTransferControllerAddress()
    Returns the Address which currently acts as the transfer controller for this call — the final call.

getTransferredCall()
    public javax.telephony.Call getTransferredCall()
    Returns the call that will be transferred.
**CiscoUnregistrationException**

**Declaration**

```java
public class CiscoUnregistrationException extends java.lang.Exception
    java.lang.Object
      |---java.lang.Throwable
        |---java.lang.Exception
          |---com.cisco.jtapi.extensions.CiscoUnregistrationException

All Implemented Interfaces

java.io.Serializable
```

**Description**

The `CiscoMediaTerminal.unregister` method throws this exception when the unregistration process fails for any reason. For example, registration would fail if the Provider were `OUT_OF_SERVICE` or if the device were already unregistered.

**See Also:**

`CiscoMediaTerminal.unregister()`

**Member Summary**

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoUnregistrationException()</td>
</tr>
<tr>
<td>CiscoUnregistrationException(java.lang.String description)</td>
</tr>
</tbody>
</table>
Class com.cisco.services.alarm

Constructors

public CiscoUnregistrationException()

ciscoUnregistrationException(String)

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
wait(), wait(), wait()

Methods inherited from class Throwable

fillInStackTrace(), get cause(), getLocalizedMessage(), getMessage(), getStackTrace(),
initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter),
printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()

Class com.cisco.services.alarm

Interfaces

Alarm

The Alarm interface is used to define Alarms in.
**Alarm Declaration**

```java
public interface Alarm
```

**All Known Implementing Classes**

- DefaultAlarm

**Description**

The Alarm interface is used to define Alarms in. An Alarm has an XML representation that it must adhere to in order to be recognized by the Alarm Service, with a DTD as shown below. An application can implement this interface or use the AlarmFactory to generate Alarms of the correct format. The Alarm is the a specification that needs to be sent to an AlarmService that will take some action based on the Alarm. Using this specification the AlarmService will access definitions available in a catalog. This catalog is maintained by the
user requiring the Alarm function to effect the appropriate action for the Alarm. The severity specified the Alarm can over-ride the severity associated with this Alarm in the catalog. If no severity is specified in the Alarm the catalog severity is used.

Alarm severities are derived from Syslog and are defined as follows:
0 = EMERGENCIES System unusable
1 = ALERTS Immediate action needed
2 = CRITICAL Critical conditions
3 = ERROR Error conditions
4 = WARNING Warning conditions
5 = NOTIFICATION Normal but significant condition
6 = INFORMATIONAL Informational messages only
7 = DEBUGGING Debugging messages

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td><strong>ALERTS</strong> static int</td>
</tr>
<tr>
<td>The application will continue working on the tasks but all functions may not be operational (one or more devices in the list are not accessible but others in the list can be accessed) Syslog severity level = 1</td>
</tr>
<tr>
<td><strong>CRITICAL</strong> static int</td>
</tr>
<tr>
<td>A critical failure, the application can not accomplish the tasks required due to this failure, eg: the app cant open the database to read the device list Syslog severity level = 2</td>
</tr>
<tr>
<td><strong>DEBUGGING</strong> static int</td>
</tr>
<tr>
<td>Very detailed information regarding errors or processing status that is only generated when DEBUG mode has been enabled Syslog severity level = 7</td>
</tr>
<tr>
<td><strong>EMERGENCIES</strong> static int</td>
</tr>
<tr>
<td>Emergency situation, a system shutdown is necessary Syslog severity level = 0</td>
</tr>
<tr>
<td><strong>ERROR</strong> static int</td>
</tr>
<tr>
<td>An error condition of some kind has occurred and the user needs to understand the nature of that failure Syslog severity level = 3</td>
</tr>
</tbody>
</table>
## Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int HIGHEST_LEVEL</td>
<td>The highest trace level, currently this is DEBUGGING with a trace level of 7</td>
</tr>
<tr>
<td>static int INFORMATIONAL</td>
<td>Information of some form not relating to errors, warnings, audit, or debug</td>
</tr>
<tr>
<td>static int LOWEST_LEVEL</td>
<td>The lowest trace level, currently this is EMERGENCIES with a trace level of 0</td>
</tr>
<tr>
<td>static int NO_SEVERITY</td>
<td>Applications can set this level to generate Alarms without a severity.</td>
</tr>
<tr>
<td>static int NOTIFICATION</td>
<td>Notification denotes a normal but significant condition</td>
</tr>
<tr>
<td>static int WARNING</td>
<td>Warning that a problem of some form exists but is not keeping the application from completing its tasks</td>
</tr>
<tr>
<td>static int NO_SEVERITY</td>
<td>Syslog severity level = 6</td>
</tr>
<tr>
<td>static int INFORMATIONAL</td>
<td>Syslog severity level = 6</td>
</tr>
<tr>
<td>static int LOWEST_LEVEL</td>
<td>Syslog severity level = 0</td>
</tr>
<tr>
<td>static int UNKNOWN_MNEMONIC</td>
<td>String used when a mnemonic is not specified during an Alarm send</td>
</tr>
<tr>
<td>static int WARNING</td>
<td>Syslog severity level = 4</td>
</tr>
</tbody>
</table>

## Methods

- `java.lang.String getFacility()`: Get the facility
- `int getSeverity()`: Get the severity
- `java.lang.String getSubFacility()`: Get the sub-facility
- `void send(java.lang.String mnemonic)`: Send the Alarm with the specified mnemonic.
- `void send(java.lang.String mnemonic, ParameterList parameterList)`: Send an Alarm with the specified mnemonic and supplied parameter list.
- `void send(java.lang.String mnemonic, java.lang.String parameterName, java.lang.String parameterValue)`: Send an Alarm with the specified mnemonic and with one parameter.

## Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static final int ALERTS</td>
<td>The application will continue working on the tasks but all functions may not be operational (one or more devices in the list are not accessible but others in</td>
</tr>
</tbody>
</table>
the list can be accessed)
Syslog severity level = 1

**CRITICAL**

```java
public static final int CRITICAL
```

A critical failure, the application can not accomplish the tasks required due to this failure, eg: the app cant open the database to read the device list
Syslog severity level = 2

**DEBUGGING**

```java
public static final int DEBUGGING
```

Very detailed information regarding errors or processing status that is only generated when DEBUG mode has been enabled
Syslog severity level = 7

**EMERGENCIES**

```java
public static final int EMERGENCIES
```

Emergency situation, a system shutdown is necessary
Syslog severity level = 0

**ERROR**

```java
public static final int ERROR
```

An error condition of some kind has occurred and the user needs to understand the nature of that failure
Syslog severity level = 3

**HIGHEST_LEVEL**

```java
public static final int HIGHEST_LEVEL
```

The highest trace level, currently this is DEBUGGING with a trace level of 7

**INFORMATIONAL**

```java
public static final int INFORMATIONAL
```

Information of some form not relating to errors, warnings, audit, or debug
Syslog severity level = 6
LOWEST_LEVEL

public static final int LOWEST_LEVEL

The lowest trace level, currently this is EMERGENCIES with a trace level of 0

NO_SEVERITY

public static final int NO_SEVERITY

Applications can set this level to generate Alarms without a severity. NOTE: This is only intended for cases where an application wants the AlarmService to use the severity associated with the Alarm in the catalog

NOTIFICATION

public static final int NOTIFICATION

Notification denotes a normal but significant condition
Syslog severity level = 5

UNKNOWN_MNEMONIC

public static final java.lang.String UNKNOWN_MNEMONIC

String used when a mnemonic is not specified during an Alarm send

WARNING

public static final int WARNING

Warning that a problem of some form exists but is not keeping the application from completing its tasks
Syslog severity level = 4

Methods

getFacility()

public java.lang.String getFacility()

Returns: the facility name of this Alarm

getSeverity()

public int getSeverity()
AlarmManager

Declaration

```java
public class AlarmManager
    extends java.lang.Object
    implements com.cisco.services.alarm.AlarmManager
```

**getSubFacility()**

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

**Returns:** the subfacility of this Alarm

**send(String)**

```java
public void send(java.lang.String mnemonic)
```

send the Alarm with the specified mnemonic. If a null or empty String is passed a mnemonic UNK is sent

**send(String, ParameterList)**

```java
public void send(java.lang.String mnemonic,
                com.cisco.services.alarm.ParameterList parameterList)
```

send an Alarm with the specified mnemonic and supplied parameter list

**send(String, String, String)**

```java
public void send(java.lang.String mnemonic,
                java.lang.String parameterName,
                java.lang.String parameterValue)
```

send an Alarm with the specified mnemonic and with one parameter
Description

The AlarmManager is used to create Alarm objects. The AlarmManager is created with a facility and AlarmService hostname and port. All alarms created by the factory will be associated with this facility. This class also maintains a reference to a single AlarmWriter that can be used system wide. An application can make use of this AlarmWriter. AlarmManager exposes a default implementation of an AlarmWriter. Applications can override this with a user defined implementation of their own AlarmWriter.

Usage: AlarmManager alarmManager = new AlarmManager(facilityName, alarmServiceHost, alarmServicePort, debugTrace, errorTrace);

Alarms are created by the factory by supplying the alarmName (mnemonic), subfacility and severity. Alarms can be cached for use in different parts of the application. During a send alarm applications can specify the the variable parameters that offer specific information to the AlarmService.

Usage:
Typically applications will maintain their own AlarmManager instance. Applications will also have to set a debug and error trace to enable the alarm tracing to also be sent to the existing trace destinations.

Setup the manager and writer classes:
AlarmWriter alarmWriter = new DefaultAlarmWriter(port, alarmServiceHost);
AlarmManager alarmManager = new AlarmManager("AA_IVR", alarmWriter, debugTrace, errorTrace);

Generating the Alarms:
create an alarm for the subfacility and a default severity.
Alarm alarm = alarmManager.createAlarm("HTTPSS", Alarm.INFORMATIONAL);
alarm.send("090T") sends the alarm with the mnemonic
alarm.send("090T", "Port is stuck", "CTIPort01") or with a mnemonic and parameter
Note: More than one parameter can be sent by specifying a ParameterList
Chapter 2  Cisco JTAPI Implementation

AlarmManager

Member Summary

Constructors

AlarmManager(java.lang.String facility, AlarmWriter writer, com.cisco.services.tracing.Trace debugTrace_, com.cisco.services.tracing.UnconditionalTrace errorTrace_)
Create an instance of the AlarmManager for the facility.

Methods

Alarm createAlarm(java.lang.String subfacility, int severity)
Creates an Alarm of required severity for the subFacility

AlarmWriter getAlarmWriter()

void setAlarmWriter(AlarmWriter writer)
Allows applications to override the AlarmWriter to be used by this AlarmManager, with a user defined AlarmWriter

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), notify(), notifyAll(), toString(), wait(), wait(), wait()

Constructors

AlarmManager(String, AlarmWriter, Trace, UnconditionalTrace)

Create an instance of the AlarmManager for the facility. Applications specify an AlarmWriter to be used by this AlarmManager to send the Alarms to the AlarmService.

**Methods**

`createAlarm(String, int)`

```java
public com.cisco.services.alarm.Alarm createAlarm(java.lang.String subfacility, int severity)
```

Creates an Alarm of required severity for the subFacility

**Returns:** an object implementing the alarm interface

`getAlarmWriter()`

```java
public com.cisco.services.alarm.AlarmWriter getAlarmWriter()
```

**Returns:** an AlarmWriter object

`setAlarmWriter(AlarmWriter)`

```java
public void setAlarmWriter(com.cisco.services.alarm.AlarmWriter writer)
```

Allows applications to override the AlarmWriter to be used by this AlarmManager, with a user defined AlarmWriter

**AlarmWriter**

**Declaration**

```java
public interface AlarmWriter
```
All Known Implementing Classes

DefaultAlarmWriter

Description

An AlarmWriter receives alarm messages and transmits it to the receiving AlarmService on a TCP link. This interface can be used to implement other AlarmWriters to be used with this implementation of com.cisco.service.alarm A DefaultAlarmWriter is provided with this implementation and can be obtained from the AlarmManager.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
</tr>
<tr>
<td>public java.lang.String getDescription()</td>
</tr>
<tr>
<td>boolean getEnabled()</td>
</tr>
<tr>
<td>java.lang.String getName()</td>
</tr>
<tr>
<td>void send(java.lang.String alarmMessage)</td>
</tr>
<tr>
<td>void setEnabled(boolean enable)</td>
</tr>
</tbody>
</table>

Methods

close()

public void close()

close the AlarmWriter

getDescription()

public java.lang.String getDescription()
Returns:  the AlarmWriter description

getEnabled()

public boolean getEnabled()

Returns:  the current enabled or disabled state of the AlarmWriter

getName()

public java.lang.String getName()

Returns:  the AlarmWriter name

send(String)

public void send(java.lang.String alarmMessage)

Send out the alarm message to the AlarmService.

Parameters:
the - Alarm to be sent

setEnabled(boolean)

public void setEnabled(boolean enable)

Enable or disable the AlarmWriter

Parameters:
enable - or disable the AlarmWriter

DefaultAlarm

Declaration

public class DefaultAlarm implements Alarm

java.lang.Object

+--com.cisco.services.alarm.DefaultAlarm
All Implemented Interfaces

Alarm

Description

An Implementation of the Alarm interface. The AlarmManager creates these Alarms when the createAlarm() method is called.

Member Summary

Constructors

DefaultAlarm(java.lang.String facility, java.lang.String subFacility, int severity, AlarmWriter alarmWriter)

Methods

java.lang.String getFacility()
int getSeverity()
java.lang.String getSubFacility()
void send(java.lang.String mnemonic)
    Send the alarm with the specified mnemonic
void send(java.lang.String mnemonic, ParameterList paramList)
    Send the alarm with the specified name and list of parameters.
void send(java.lang.String mnemonic, java.lang.String paramName, java.lang.String paramValue)
    Send the alarm with the specified name and parameter

Inherited Member Summary

Fields inherited from interface Alarm

ALERTS, CRITICAL, DEBUGGING, EMERGENCIES, ERROR, HIGHEST_LEVEL, INFORMATIONAL, LOWEST_LEVEL, NOTIFICATION, NO_SEVERITY, UNKNOWN_MNEMONIC, WARNING
Constructors

DefaultAlarm(String, String, int, AlarmWriter)

public DefaultAlarm(java.lang.String facility,
java.lang.String subFacility, int severity,
com.cisco.services.alarm.AlarmWriter alarmWriter)

Methods

getFacility()

public java.lang.String getFacility()

Specified By: getFacility in interface Alarm

getSeverity()

public int getSeverity()

Specified By: getSeverity in interface Alarm

getSubFacility()

public java.lang.String getSubFacility()

Specified By: getSubFacility in interface Alarm

send(String)

public void send(java.lang.String mnemonic)

Send the alarm with the specified mnemonic

Specified By: send in interface Alarm

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
toString(), wait(), wait(), wait()
DefaultAlarmWriter

Declaration

```java
public class DefaultAlarmWriter implements AlarmWriter
```

All Implemented Interfaces

```java
AlarmWriter
```
DefaultAlarmWriter implementation of the AlarmWriter interface. DefaultAlarmWriter maintains a queue of a fixed size to which the alarms are written. The sending of the alarms to the alarm service takes place on a separate thread. The queue is fixed size.

**Member Summary**

### Constructors

- DefaultAlarmWriter(int port, java.lang.String alarmServiceName)
  Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and a queue size of fifty (50).
- DefaultAlarmWriter(int port, java.lang.String alarmServiceName, int queueSize)
  Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and queue size.
- DefaultAlarmWriter(int port, java.lang.String alarmServiceName, int queueSize, com.cisco.services.tracing.ConditionalTrace debugTrace_, com.cisco.services.tracing.UnconditionalTrace errorTrace_)
  Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and queue size.

### Methods

- void close()
  Shutdown the send thread and close the socket
- java.lang.String getDescription()
- boolean getEnabled()
- java.lang.String getName()
- static void main(java.lang.String args)
- void send(java.lang.String alarmMessage)
  send the Alarm to the alarm service
- void setEnabled(boolean enable)
  Applications can dynamically enable or disable the AlarmWriter
Constructors

DefaultAlarmWriter(int, String)

public DefaultAlarmWriter(int port,
java.lang.String alarmServiceName)
throws UnknownHostException

Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and a queue size of fifty (50). The AlarmService is listening on this port for Alarm messages.

Parameters:
port - port on which the alarm service is listening
alarmServiceName - The host name of the machine with the Alarm service

Throws:
java.net.UnknownHostException

DefaultAlarmWriter(int, String, int)

public DefaultAlarmWriter(int port,
java.lang.String alarmServiceName, int queueSize)
throws UnknownHostException

Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and queue size. The AlarmService is listening on this port for Alarm messages.

Parameters:
port - port on which the alarm service is listening
alarmServiceName - The host name of the machine with the Alarm service

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()
queueSize - the size of the queue to be maintained in the alarm writer

Throws:
java.net.UnknownHostException

DefaultAlarmWriter(int port, java.lang.String alarmServiceName, int queueSize, com.cisco.services.tracing.ConditionalTrace debugTrace_, com.cisco.services.tracing.UnconditionalTrace errorTrace_)

Constructor for the DefaultAlarmWriter which takes the AlarmService hostname, port and queue size. The AlarmService is listening on this port for Alarm messages.

Parameters:
port - port on which the alarm service is listening
alarmServiceName - The host name of the machine with the Alarm service
queueSize - the size of the queue to be maintained in the alarm writer

Throws:
java.net.UnknownHostException

Methods

close()
public void close()

Shutdown the send thread and close the socket

Specified By: close in interface AlarmWriter

description()
public java.lang.String description()

Specified By: description in interface AlarmWriter
ParameterList

Declaration

public class ParameterList

java.lang.Object
ParameterList is a list of name value pairs that is used to send additional (and optional) user defined parameters to the AlarmService. These parameters can contain the specifics of an Alarm. As an example, a LowResourceAlarm can have and parameter that informs the service which particular resource is low:

name="CPUUsage"
value="0.9"

These parameters are user definable but must, however, also be pre-defined in the AlarmService catalog.

### Member Summary

#### Constructors

- `ParameterList()`
  - Default constructor for the ParameterList
- `ParameterList(java.lang.String name, java.lang.String value)`
  - Constructor that takes a name value pair.

#### Methods

- `void addParameter(java.lang.String name, java.lang.String value)`
  - method used to add additional name value pairs (parameters) to the list
- `java.lang.String[] getParameterNames()`
  - Get the parameter names in the list
- `java.lang.String getParameterValue(java.lang.String parameterName)`
  - get the value for a parameter
- `void removeAllParameters()`
  - remove all the parameters in the list
- `void removeParameter(java.lang.String parameterName)`
  - remove a particular parameter if it is in the list
- `java.lang.String toString()`
Chapter 2  Cisco JTAPI Implementation

ParameterList

### Constructors

**ParameterList()**

```java
public ParameterList()
```

Default constructor for the ParameterList

**ParameterList(String, String)**

```java
public ParameterList(java.lang.String name,
java.lang.String value)
```

Constructor that takes a name value pair.

### Methods

**addParameter(String, String)**

```java
public void addParameter(java.lang.String name,
java.lang.String value)
```

method used to add additional name value pairs (parameters) to the list

**getParameterNames()**

```java
public java.lang.String[] getParameterNames()
```

Get the parameter names in the list

**getParameters(String)**

```java
public java.lang.String getParameterValue(java.lang.String
parameterName)
```

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
wait(), wait(), wait()
### Class com.cisco.services.tracing

**get the value for a parameter**

**Returns:** value of a parameter

**removeAllParameters()**

```java
public void removeAllParameters()
```

remove all the parameters in the list

**removeParameter(String)**

```java
public void removeParameter(java.lang.String parameterName)
```

remove a particular parameter if it is in the list

**toString()**

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

**Overrides:** toString in class Object

---

### Class Summary

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConditionalTrace</td>
<td>The ConditionalTrace interface extends the Trace interface and defines the methods that allow enabling and disabling of tracing for this particular condition.</td>
</tr>
<tr>
<td>Trace</td>
<td>The Trace interface defines the methods that allow application tracing.</td>
</tr>
<tr>
<td>TraceManager</td>
<td>The TraceManager interface defines the methods that allow applications trace management.</td>
</tr>
<tr>
<td>TraceModule</td>
<td>The TraceModule interface serves two purposes.</td>
</tr>
<tr>
<td>TraceWriter</td>
<td>TraceWriterManager contains the list of TraceWriter objects that are used to implement the tracing.</td>
</tr>
<tr>
<td>TraceWriterManager</td>
<td>Introduction</td>
</tr>
</tbody>
</table>
Chapter 2  Cisco JTAPI Implementation

BaseTraceWriter

Class Summary

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UnconditionalTrace</td>
<td>The UnconditionalTrace interface extends the Trace interface.</td>
</tr>
<tr>
<td>BaseTraceWriter</td>
<td>This abstract class is useful for supplying a default, non-printing TraceWriter to a TraceWriterManager. This class must be extended to provide the functionality to trace to different streams.</td>
</tr>
<tr>
<td>ConsoleTraceWriter</td>
<td>Supplies a console TraceWriter to trace to System.out.</td>
</tr>
<tr>
<td>LogFileTraceWriter</td>
<td>OutputStreamTraceWriter wraps an output stream in a TraceWriter.</td>
</tr>
<tr>
<td>SyslogTraceWriter</td>
<td>SyslogTraceWriter refines the BaseTraceWriter to allow tracing to syslog.</td>
</tr>
<tr>
<td>TraceManagerFactory</td>
<td>The TraceManagerFactory class is a class by which applications obtain a TraceManager object.</td>
</tr>
</tbody>
</table>

BaseTraceWriter

Declaration

```java
public abstract class BaseTraceWriter implements TraceWriter
```

All Implemented Interfaces

- TraceWriter
Direct Known Subclasses

- ConsoleTraceWriter
- LogFileTraceWriter
- OutputStreamTraceWriter
- SyslogTraceWriter

Description

This abstract class is useful for supplying a default, non-printing TraceWriter to a TraceWriterManager. This class must be extended to provide the functionality to trace to different streams. The doPrintln() method must be implemented by the extending class.

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected BaseTraceWriter(int[] traceLevels, java.lang.String name, java.lang.String description)</td>
</tr>
<tr>
<td>BaseTraceWriter with trace levels as passed in traceLevels in the array falling outside the range Trace.LOWEST_LEVEL and Trace.HIGHEST_LEVEL are ignored</td>
</tr>
<tr>
<td>protected BaseTraceWriter(int maxTraceLevel, java.lang.String name, java.lang.String description)</td>
</tr>
<tr>
<td>BaseTraceWriter that traces all levels up to the maxTraceLevel. The trace level is maintained in the range [Trace.HIGHEST_LEVEL, Trace.LOWEST_LEVEL]</td>
</tr>
<tr>
<td>protected BaseTraceWriter(java.lang.String name, java.lang.String description)</td>
</tr>
<tr>
<td>BaseTraceWriter which only traces the lowest level (severity level, Trace.LOWEST_LEVEL) messages</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
</tr>
<tr>
<td>protected void doClose()</td>
</tr>
<tr>
<td>protected void doFlush()</td>
</tr>
<tr>
<td>protected abstract void doPrintln(java.lang.String message, int messageNumber)</td>
</tr>
<tr>
<td>Must be implemented by the various TraceWriters extending BaseTraceWriter to provide the specific tracing functionality</td>
</tr>
<tr>
<td>void flush()</td>
</tr>
<tr>
<td>java.lang.String getDescription()</td>
</tr>
<tr>
<td>boolean getEnabled()</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

BaseTraceWriter

Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>getName()</td>
</tr>
<tr>
<td>int[]</td>
<td>getTraceLevels()</td>
</tr>
<tr>
<td>void</td>
<td>println(java.lang.String, int severity)</td>
</tr>
<tr>
<td>void</td>
<td>setTraceLevels(int[] levels)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>toString()</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
wait(), wait(), wait()

Constructors

BaseTraceWriter(int[], String, String)

protected BaseTraceWriter(int[] traceLevels,
                          java.lang.String name,
                          java.lang.String description)

BaseTraceWriter with trace levels as passed in traceLevels in the array
falling outside the range Trace.LOWEST_LEVEL and
Trace.HIGHEST_LEVEL are ignored

Parameters:

traceLevels - array of trace levels

See Also:

Trace

BaseTraceWriter(int, String, String)

protected BaseTraceWriter(int maxTraceLevel,
                          java.lang.String name,
                          java.lang.String description)
BaseTraceWriter that traces all levels upto the maxTraceLevel The trace level is maintained in the range [Trace.HIGHEST_LEVEL, Trace.LOWEST_LEVEL ]

Parameters:

See Also:

BaseTraceWriter(String, String)

protected BaseTraceWriter(java.lang.String name, java.lang.String description)

BaseTraceWriter which only traces the lowest level ie severity level, Trace.LOWEST_LEVEL messages

Parameters:

See Also:

Methods

close()

public final void close()

Description copied from interface: com.cisco.services.tracing.TraceWriter

Releases any resources associated by this TraceWriter.

Specified By: close in interface TraceWriter

doClose()

protected void doClose()

doFlush()

protected void doFlush()
Chapter 2      Cisco JTAPI Implementation

BaseTraceWriter

protected abstract void doPrintln(String message, int messageNumber)

Must be implemented by the various TraceWriters extending BaseTraceWriter to provide the specific tracing functionality

dflush()

public final void flush()

Description copied from interface: com.cisco.services.tracing.TraceWriter

Specified By: flush in interface TraceWriter

getDescription()

public final java.lang.String getDescription()

Specified By: getDescription in interface TraceWriter

getEnabled()

public boolean getEnabled()

Description copied from interface: com.cisco.services.tracing.TraceWriter

Returns whether the println method will print anything or not. A closed TraceWriter will always return false from this method.

Specified By: getEnabled in interface TraceWriter

getName()

public final java.lang.String getName()

Specified By: getName in interface TraceWriter

getTraceLevels()

public final int[] getTraceLevels()
ConditionalTrace

**Declaration**

```java
public interface ConditionalTrace extends Trace
```

**Specification By:** `getTraceLevels` in interface `TraceWriter`

```java
public final void println(String message, int severity)
```

*Description copied from interface:* `com.cisco.services.tracing.TraceWriter`

Prints the specified string followed by a carriage return. The concrete `TraceWriter` class will use the severity to block out messages from a particular stream. Each trace writer has a notion of the highest level trace it traces.

**Specification By:** `println` in interface `TraceWriter`

**Specification By:** `setTraceLevels` in interface `TraceWriter`

```java
public final void setTraceLevels(int[] levels)
```

*Description copied from interface:* `com.cisco.services.tracing.TraceWriter`

Sets the trace levels that will be traced by this `TraceWriter`.

**Specification By:** `setTraceLevels` in interface `TraceWriter`

```java
@Override
public final java.lang.String toString()
```

*Overrides:* `toString` in class `Object`
Chapter 2      Cisco JTAPI Implementation

ConditionalTrace

All Superinterfaces

Trace

Description

The ConditionalTrace interface extends the Trace interface and defines the methods that allow enabling and disabling of tracing for this particular condition.

Typically, applications would obtain one ConditionalTrace object per each condition that they need to trace under certain circumstances but not always (e.g. AUDIT, INFO, etc.).

Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void disable()</td>
<td>Disables this condition for tracing.</td>
</tr>
<tr>
<td>void enable()</td>
<td>Enables this condition for tracing.</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Fields inherited from interface Trace

ALERTS, ALERTS_TRACE_NAME, CRITICAL, CRITICAL_TRACE_NAME, DEBUGGING, DEBUGGING_TRACE_NAME, EMERGENCIES, EMERGENCIES_TRACE_NAME, ERROR, ERROR_TRACE_NAME, HIGHEST_LEVEL, INFORMATIONAL, INFORMATIONAL_TRACE_NAME, LOWEST_LEVEL, NOTIFICATION, NOTIFICATION_TRACE_NAME, WARNING, WARNING_TRACE_NAME

Methods inherited from interface Trace
ConsoleTraceWriter

Methods

disable()
public void disable()
Disables this condition for tracing.

enable()
public void enable()
Enables this condition for tracing.

Declaration

public final class ConsoleTraceWriter extends BaseTraceWriter

java.lang.Object
|-- com.cisco.services.tracing.BaseTraceWriter
|   |-- com.cisco.services.tracing.ConsoleTraceWriter

AllImplemented Interfaces

TraceWriter
**Description**

Supplies a console TraceWriter to trace to System.out.

**See Also:**

*Trace*

---

### Member Summary

#### Constructors

- `ConsoleTraceWriter()`: Default constructor, traces all severity levels
- `ConsoleTraceWriter(int maxTraceLevel)`: Constructor that sets the maximum level to be traced.
- `ConsoleTraceWriter(int[] traceLevels)`: Construct a ConsoleTraceWriter with an array of trace levels. Only traces with the severity in the tracelevel array are traced.

#### Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>protected void doFlush()</code></td>
<td></td>
</tr>
<tr>
<td><code>protected void doPrintln(java.lang.String message, int messageNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>static void main(java.lang.String args)</code></td>
<td></td>
</tr>
</tbody>
</table>

---

### Inherited Member Summary

**Methods inherited from class** `BaseTraceWriter`

- `close()`, `doClose()`, `flush()`, `getDescription()`, `getEnabled()`, `getName()`, `getTraceLevels()`, `println(String, int)`, `setTraceLevels(int[])`, `toString()`

**Methods inherited from class** `Object`

- `clone()`, `equals(Object)`, `finalize()`, `getClass()`, `hashCode()`, `notify()`, `notifyAll()`, `wait()`, `wait()`, `wait()`
Constructors

ConsoleTraceWriter()

public ConsoleTraceWriter()

Default constructor, traces all severity levels

ConsoleTraceWriter(int)

public ConsoleTraceWriter(int maxTraceLevel)

Constructor that sets the maximum level to be traced.

Parameters:

See Also:

Trace

ConsoleTraceWriter(int[])

public ConsoleTraceWriter(int[] traceLevels)

Construct a ConsoleTraceWriter with an array of trace levels Only traces with the severity in the tracelevel array are traced

Parameters:

int[] - traceLevels

See Also:

Trace

Methods

doFlush()

protected final void doFlush()

Overrides: doFlush in class BaseTraceWriter

doPrintln(String, int)

protected final void doPrintln(java.lang.String message, int messageNumber)
LogFileTraceWriter

Description copied from class: com.cisco.services.tracing.BaseTraceWriter

Must be implemented by the various TraceWriters extending BaseTraceWriter to provide the specific tracing functionality

Overrides: doPrintln in class BaseTraceWriter

main(String[])

public static void main(java.lang.String[] args)

LogFileTraceWriter

Declaration

public final class LogFileTraceWriter extends BaseTraceWriter

java.lang.Object
| ^--com.cisco.services.tracing.BaseTraceWriter
| ^--com.cisco.services.tracing.LogFileTraceWriter

All Implemented Interfaces

TraceWriter

Description

This class extends the BaseTraceWriter class to implement a TraceWriter that writes to a set of log files, rotating among them as each becomes filled to a specified capacity and stores them in a specified directory.

Each of the log files is named according to a pattern controlled by three properties, CurrentFile, FileNameBase, and FileExtension. The CurrentFile property determines which log file, by ordinal number, is being
written at present, the FileNameBase property determines the prefix of each log file name, and the FileExtension property determines the suffix, e.g. “txt”. From these properties, log files are named FileNameBase
LeadingZeroPadding CurrentFile.FileExtension. The CurrentFile property takes on a value from 1 to the value of the MaxFiles property. Note that the CurrentFile property, when converted to a String, is padded with leading zeroes depending on the values of the MaxFiles and CurrentFile properties. An index file tracks the index of the last file written. If the logFileWriter is recreated (for example if an application is restarted) new files will continue from the last written index.

Where the log files are stored is determined by the path, dirNameBase, useSameDir. If a path is not specified, the current path is used as default. If a dirNameBase is not specified, it write log files in the path. Depending upon whether useSameDir is true or false, files are written to the same directory or a new directory, each time an instance of LogFileTraceWriter is created. In case new directories are being made each time, the directory name will consist of the dirNameBase and a number, seperated by an ‘_’. The number is one more than the greatest number associated with directories with the same dirNameBase in the path. While specifying the path, you may use either a “/” or “\", but not “\”.

The LogFileTraceWriter keeps track of how many bytes have been written to the current log file. When that number grows within approximately LogFileTraceWriter.ROLLOVER_THRESHOLD bytes, tracing continues to the next file, which is either CurrentFile + 1 if CurrentFile is not equal to MaxFiles, or 1 if CurrentFile is equal to MaxFiles.

**Note:** All properties of this class are specified in the constructor; there is no way to change them dynamically. **Caveat:** If two instances of LogFileTraceWriter are created with the same path and dirNameBase, and useSameDir is true, they may write to the same file.

**Example**

The following code instantiates a LogFileTraceWriter that will create log files called “MyLog01.log” through “MyLog12.log”. Each file will grow to approximately 100K bytes in size before the next file is created:

```java
LogFileTraceWriter out = new LogFileTraceWriter ( "MyLog", "log", 12, 100 * 1024 );
```

will create a log
LogFileTraceWriter which will rotate traces to 12 files from Mylog01.log and Mylog12.log with a file size of 100 KBytes. By default the tracing is set to the HIGHEST_LEVEL

Example
The following code constructs a LogFileTraceWriter which stores the log files in the path “c:/LogFiles” in a sub directory, “Run”. The files will be named MyLogXX.log. The number of rotating files will be 12 with a size of 100 KB. The same directory is used for each instance of the application.

LogFileTraceWriter out = new LogFileTraceWriter ( “c:/logFiles”, “Run”, “MyLog”, “log”, 12, 100*1024, true );

See Also:
Trace

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fields</strong></td>
</tr>
<tr>
<td>static java.lang.String DEFAULT_FILE_NAME_BASE</td>
</tr>
<tr>
<td>static java.lang.String DEFAULT_FILE_NAME_EXTENSION</td>
</tr>
<tr>
<td>static char DIR_BASE_NAME_NUM_SEPERATOR</td>
</tr>
<tr>
<td>static int MIN_FILE_SIZE</td>
</tr>
<tr>
<td>static int MIN_FILES</td>
</tr>
<tr>
<td>static int ROLLOVER_THRESHOLD</td>
</tr>
<tr>
<td><strong>Constructors</strong></td>
</tr>
<tr>
<td>LogFileTraceWriter(java.lang.String fileNameBase, java.lang.String fileNameExtension, int maxFiles, int maxFileSize)</td>
</tr>
</tbody>
</table>
LogFileTraceWriter

Default constructor for LogFileTraceWriter that rotates among an arbitrary number of files with tracing for all levels.

LogFileTraceWriter(java.lang.String path, java.lang.String dirNameBase, java.lang.String fileNameBase, java.lang.String fileNameExtension, int maxFiles, int maxFileSize, int maxTraceLevel, boolean useSameDir)

Constructs a LogFileTraceWriter that rotates among an arbitrary number of files storing them in a specified directory.

Methods

protected void doClose()
Closes this OutputStream.

protected void doFlush()

protected void doPrintln(java.lang.String message, int messageNumber)

int getCurrentFile()
Returns the CurrentFile property

java.lang.String getFileExtension()
Returns the FileExtension property

java.lang.String getFileNameBase()
Returns the FileNameBase property

java.lang.String getHeader()
Get the header string that will be written at the beginning of each log file.

int getMaxFiles()
Returns the MaxFiles property

int getMaxFileSize()
Returns the MaxFileSize property

void setHeader(java.lang.String header)
Set the constant header string that will be written at the beginning of every file, trace writing continues from the next line after the header is written.
Inherited Member Summary

Methods inherited from class BaseTraceWriter:

- close()
- flush()
- getDescription()
- getEnabled()
- getName()
- getTraceLevels()
- println(String, int)
- setTraceLevels(int[])
- toString()

Methods inherited from class Object:

- clone()
- equals(Object)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- wait()
- wait()
- wait()

Fields

**DEFAULT_FILE_NAME_BASE**

public static final java.lang.String DEFAULT_FILE_NAME_BASE

**DEFAULT_FILE_NAME_EXTENSION**

public static final java.lang.String DEFAULT_FILE_NAME_EXTENSION

**DIR_BASE_NAME_NUM_SEPERATOR**

public static final char DIR_BASE_NAME_NUM_SEPERATOR

**MIN_FILE_SIZE**

public static final int MIN_FILE_SIZE

**MIN_FILES**

public static final int MIN_FILES

**ROLLOVER_THRESHOLD**

public static final int ROLLOVER_THRESHOLD
Constructors

LogFileTraceWriter(String, String, int, int)

public LogFileTraceWriter(String fileNameBase,
                          String fileNameExtension, int maxFiles,
                          int maxFileSize)
          throws IOException

Default constructor for LogFileTraceWriter that rotates among an arbitrary number of files with tracing for all levels. Since a path and Directory Base name is not specified, it writes the files to the current directory without any sub directories.

Parameters:

Throws:

java.io.IOException

LogFileTraceWriter(String, String, String, String, int, int, boolean)

public LogFileTraceWriter(String path,
                          String dirNameBase,
                          String fileNameBase,
                          String fileNameExtension, int maxFiles,
                          int maxFileSize, boolean useSameDir)
          throws IOException

Default constructor for LogFileTraceWriter that rotates among an arbitrary number of files with tracing for all levels.

Parameters:

Throws:

java.io.IOException

LogFileTraceWriter(String, String, String, String, int, int, int, boolean)

public LogFileTraceWriter(String path,
                          String dirNameBase,
                          String fileNameBase,
                          String fileNameExtension, int maxFiles,
                          int maxFileSize, int maxTraceLevel,
                          boolean useSameDir)
          throws IOException
LogFileTraceWriter

Constructs a LogFileTraceWriter that rotates among an arbitrary number of files storing them in a specified directory.

**Parameters:**

**Throws:**

java.io.IOException

**Methods**

**doClose()**

protected void doClose()

Closes this OutputStream. Any log file that is currently open will be closed as well.

**Overrides:** doClose in class BaseTraceWriter

**doFlush()**

protected void doFlush()

**Overrides:** doFlush in class BaseTraceWriter

**doPrintln(String, int)**

protected void doPrintln(java.lang.String message, int messageNumber)

**Description copied from class:** com.cisco.services.tracing.BaseTraceWriter

Must be implemented by the various TraceWriters extending BaseTraceWriter to provide the specific tracing functionality

**Overrides:** doPrintln in class BaseTraceWriter

**getCurrentFile()**

public int getCurrentFile()

Returns the CurrentFile property

**Returns:** the CurrentFile property
LogFileTraceWriter

getFileExtension()
public java.lang.String getFileExtension()

Returns the FileExtension property

Returns: the FileExtension property

getFileNameBase()
public java.lang.String getFileNameBase()

Returns the FileNameBase property

Returns: the FileNameBase property

getHeader()
public java.lang.String getHeader()

Get the header string that will be written at the beginning of each log file.

Returns: the Header Property

getMaxFiles()
public int getMaxFiles()

Returns the MaxFiles property

Returns: the MaxFiles property

getMaxFileSize()
public int getMaxFileSize()

Returns the MaxFileSize property

Returns: the MaxFileSize property

setHeader(String)
public void setHeader(java.lang.String header)

Set the constant header string that will be written at the beginning of every file, trace writing continues from the next line after the header is written. If setHeader is called after a file output has started, it will take effect from the next file to be written.

Usage:
Chapter 2 Cisco JTAPI Implementation

OutputStreamTraceWriter

Declaration

```java
public final class OutputStreamTraceWriter extends BaseTraceWriter
```

All Implemented Interfaces

- `TraceWriter`

Description

OutputStreamTraceWriter wraps an output stream in a TraceWriter. This simplifies adding custom tracing classes that can co-exist with other TraceWriters.

Member Summary

<table>
<thead>
<tr>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream)</td>
</tr>
<tr>
<td>Default constructor which is aut flushing</td>
</tr>
</tbody>
</table>
OutputStreamTraceWriter

Constructors

OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream)  
Public OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream)  
Default constructor which is autoflushing

See Also:

Trace

Member Summary

OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream, boolean autoFlush)  
Create an OutputStreamTraceWriter

Constructors

OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream)  
Public OutputStreamTraceWriter(int maxTraceLevel, java.io.OutputStream outputStream)  
Default constructor which is autoflushing

See Also:

Trace

Inherited Member Summary

Methods inherited from class BaseTraceWriter

close(), flush(), getDescription(), getEnabled(), getName(), getTraceLevels(), println(String, int), setTraceLevels(int[]), toString()

Methods inherited from class Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(), wait(), wait()
OutputStreamTraceWriter(int, OutputStream, boolean)

public OutputStreamTraceWriter(int maxTraceLevel,
                              java.io.OutputStream outputStream,
                              boolean autoFlush)

Create an OutputStreamTraceWriter

Parameters:

See Also:

Trace

Methods

doClose()

protected void doClose()

Overrides: doClose in class BaseTraceWriter

doFlush()

protected void doFlush()

Overrides: doFlush in class BaseTraceWriter

doPrintln(String, int)

protected void doPrintln(java.lang.String message,
                         int messageNumber)

Description copied from class:

com.cisco.services.tracing.BaseTraceWriter

Must be implemented by the various TraceWriters extending
BaseTraceWriter to provide the specific tracing functionality

Overrides: doPrintln in class BaseTraceWriter

getOutputStream()

public java.io.OutputStream getOutputStream()

Returns: the output stream associated with the TraceWriter
SyslogTraceWriter

Declaration

```java
public final class SyslogTraceWriter extends BaseTraceWriter
```

```
java.lang.Object
| +-- com.cisco.services.tracing.BaseTraceWriter
|    +-- com.cisco.services.tracing.SyslogTraceWriter
```

All Implemented Interfaces

```java
TraceWriter
```

Description

SyslogTraceWriter refines the BaseTraceWriter to allow tracing to syslog. Cisco syslog specification calls for sending low level traces to a syslog collector in the form of UDP messages. No buffering is done in this TraceWriter. The SyslogTraceWriter makes an exception to the println() method in that it places a ‘\0’ instead of a System specified line separator to terminate the message packet.

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constructors</strong></td>
</tr>
<tr>
<td>SyslogTraceWriter(int port, java.lang.String collector)</td>
</tr>
<tr>
<td>Default SyslogTraceWriter with a max trace level of INFORMATIONAL</td>
</tr>
<tr>
<td>SyslogTraceWriter(int port, java.lang.String collector, int maxTraceLevel)</td>
</tr>
<tr>
<td>SyslogTraceWriter with max trace level specified</td>
</tr>
</tbody>
</table>
Chapter 2      Cisco JTAPI Implementation

SyslogTraceWriter

### Member Summary

**SyslogTraceWriter**

- **SyslogTraceWriter(int port, java.lang.String collector, int[] traceLevels)**
  - SyslogTraceWriter which takes an array of trace levels.

**Methods**

- **void doClose()**
  - Closes the socket

- **protected void doPrintln(java.lang.String message, int messageNumber)**
  - The SyslogTraceWriter makes an exception to the println() method in that it places a \0 instead of a System specified line separator to terminate the message packet.

- **static void main(java.lang.String args)**

### Inherited Member Summary

**Methods inherited from class** [BaseTraceWriter](#)

- close(), doFlush(), flush(), getDescription(), getEnabled(), getName(), getTraceLevels(), println(String, int), setTraceLevels(int[]), toString()

**Methods inherited from class** [Object](#)

- clone(), equals(Object), finalize(), getClass(), notify(), notifyAll(), wait(), wait(), wait()

### Constructors

**SyslogTraceWriter(int, String)**

- **public SyslogTraceWriter(int port, java.lang.String collector)**
  - Default SyslogTraceWriter with a max trace level of INFORMATIONAL

**See Also:**

- [Trace](#)
SyslogTraceWriter(int, String, int)

public SyslogTraceWriter(int port, java.lang.String collector,
                          int maxTraceLevel)

SyslogTraceWriter with max trace level specified

See Also:
       Trace

SyslogTraceWriter(int, String, int[])

public SyslogTraceWriter(int port, java.lang.String collector,
                          int[] traceLevels)

SyslogTraceWriter which takes an array of trace levels.

See Also:
       Trace

Methods

docClose()

public void docClose()

Closes the socket

Overrides: docClose in class BaseTraceWriter

doPrintln(String, int)

protected void doPrintln(java.lang.String message,
                          int messageNumber)

The SyslogTraceWriter makes an exception to the println() method in
that it places a '0' instead of a System specified line separator to terminate
the message packet. The portion of the message after a '
' or 'r' is ignored

Overrides: doPrintln in class BaseTraceWriter

dmain(String[])

public static void main(java.lang.String[] args)
Trace

Declaration

public interface Trace

All Known Subinterfaces

ConditionalTrace, UnconditionalTrace

Description

The Trace interface defines the methods that allow application tracing. Trace also defines the standard trace types as specified by Syslog Trace Logging. Syslog currently defines 8 levels of trace. The severity of the message is indicated in the trace as a number ranging between [0-7] (0 and 7 included). Currently 7 is HIGHEST_LEVEL and 0 is the LOWEST_LEVEL trace. All 8 levels are predefined here as static int types for reference in tracing sub-system implementations.

The severities traced are as follows:
0 = EMERGENCIES System unusable
1 = ALERTS Immediate action needed
2 = CRITICAL Critical conditions
3 = ERROR Error conditions
4 = WARNING Warning conditions
5 = NOTIFICATION Normal but significant condition
6 = INFORMATIONAL Informational messages only
7 = DEBUGGING Debugging messages
Member Summary

Fields

static int ALERTS
  The application will continue working on the tasks but all functions may not be operational (one or more devices in the list are not accessible but others in the list can be accessed)
  Syslog severity level = 1
static java.lang.String ALERTS_TRACE_NAME
  String descriptor for ALERTS trace level
static int CRITICAL
  A critical failure, the application can not accomplish the tasks required due to this failure, eg: the app cant open the database to read the device list
  Syslog severity level = 2
static java.lang.String CRITICAL_TRACE_NAME
  String descriptor for CRITICAL trace level
static int DEBUGGING
  Very detailed information regarding errors or processing status that is only generated when DEBUG mode has been enabled
  Syslog severity level = 7
static java.lang.String DEBUGGING_TRACE_NAME
  String descriptor for the DEBUGGING trace level
static int EMERGENCIES
  Emergency situation, a system shutdown is necessary
  Syslog severity level = 0
static java.lang.String EMERGENCIES_TRACE_NAME
  String descriptor for EMERGENCIES trace level
static int ERROR
  An error condition of some kind has occurred and the user needs to understand the nature of that failure
  Syslog severity level = 3
static java.lang.String ERROR_TRACE_NAME
  String descriptor for ERROR trace level
static int HIGHEST_LEVEL
  The highest trace level, currently this is DEBUGGING with a trace level of 7
static java.lang.String INFORMATIONAL_TRACE_NAME
  String descriptor for INFORMATIONAL trace level
static int INFORMATIONAL
  Information of some form not relating to errors, warnings, audit, or debug
  Syslog severity level = 6
static java.lang.String INFORMATIONAL_TRACE_NAME
  String descriptor for INFORMATIONAL trace level
static int LOWEST_LEVEL
  The lowest trace level, currently this is EMERGENCIES with a trace level of 0
  Syslog severity level = 0
### Member Summary

<table>
<thead>
<tr>
<th>Static Int</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTIFICATION</strong></td>
<td>Notification denotes a normal but significant condition. Syslog severity level = 5</td>
</tr>
<tr>
<td><strong>WARNING</strong></td>
<td>Warning that a problem of some form exists but is not keeping the application from completing its tasks. Syslog severity level = 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTIFICATION_TRACE_NAME</strong></td>
<td>String descriptor for NOTIFICATION trace level</td>
</tr>
<tr>
<td><strong>WARNING_TRACE_NAME</strong></td>
<td>String descriptor for WARNING trace level</td>
</tr>
</tbody>
</table>

### Methods

- **java.lang.String getName()**
  - Returns the name of this Trace object.
- **java.lang.String getSubFacility()**
  - Returns the subFacility of trace.
- **int getType()**
  - Returns the type of trace.
- **boolean isEnabled()**
  - Returns the state of this Trace object.
- **void println(java.lang.Object object)**
  - Prints the string returned by the Object.toString() method and terminates the line as defined by the system.
- **void println(java.lang.String message)**
  - Prints a message in the same format as Trace.print() and terminates the line as defined by the system.
- **void println(java.lang.String mnemonic, java.lang.Object object)**
  - Prints the string returned by the Object.toString() method and terminates the line as defined by the system.
- **void println(java.lang.String mnemonic, java.lang.String message)**
  - Prints a message in the same format as Trace.print() and terminates the line as defined by the system.
- **void setDefaultMnemonic(java.lang.String mnemonic)**
  - Sets a default mnemonic for all messages printed out to this trace.
Fields

**ALERTS**

```java
public static final int ALERTS
```

The application will continue working on the tasks but all functions may not be operational (one or more devices in the list are not accessible but others in the list can be accessed)
Syslog severity level = 1

**ALERTS_TRACE_NAME**

```java
public static final java.lang.String ALERTS_TRACE_NAME
```

String descriptor for ALERTS trace level

**CRITICAL**

```java
public static final int CRITICAL
```

A critical failure, the application can not accomplish the tasks required due to this failure, eg: the app cant open the database to read the device list
Syslog severity level = 2

**CRITICAL_TRACE_NAME**

```java
public static final java.lang.String CRITICAL_TRACE_NAME
```

String descriptor for CRITICAL trace level

**DEBUGGING**

```java
public static final int DEBUGGING
```

Very detailed information regarding errors or processing status that is only generated when DEBUG mode has been enabled
Syslog severity level = 7

**DEBUGGING_TRACE_NAME**

```java
public static final java.lang.String DEBUGGING_TRACE_NAME
```

String descriptor for the DEBUGGING trace level

**EMERGENCIES**

```java
public static final int EMERGENCIES
```
Emergency situation, a system shutdown is necessary
Syslog severity level = 0

**EMERGENCIES_TRACE_NAME**

```java
public static final java.lang.String EMERGENCIES_TRACE_NAME
```

String descriptor for EMERGENCIES trace level

**ERROR**

```java
public static final int ERROR
```

An error condition of some kind has occurred and the user needs to understand the nature of that failure
Syslog severity level = 3

**ERROR_TRACE_NAME**

```java
public static final java.lang.String ERROR_TRACE_NAME
```

String descriptor for ERROR trace level

**HIGHEST_LEVEL**

```java
public static final int HIGHEST_LEVEL
```

The highest trace level, currently this is DEBUGGING with a trace level of 7

**INFORMATIONAL**

```java
public static final int INFORMATIONAL
```

Information of some form not relating to errors, warnings, audit, or debug
Syslog severity level = 6

**INFORMATIONAL_TRACE_NAME**

```java
public static final java.lang.String INFORMATIONAL_TRACE_NAME
```

String descriptor for INFORMATIONAL trace level

**LOWEST_LEVEL**

```java
public static final int LOWEST_LEVEL
```

String descriptor for LOWEST_LEVEL
The lowest trace level, currently this is EMERGENCIES with a trace level of 0

**NOTIFICATION**

```java
public static final int NOTIFICATION
```

Notification denotes a normal but significant condition
Syslog severity level = 5

**NOTIFICATION_TRACE_NAME**

```java
public static final java.lang.String NOTIFICATION_TRACE_NAME
```

String descriptor for NOTIFICATION trace level

**WARNING**

```java
public static final int WARNING
```

Warning that a problem of some form exists but is not keeping the application from completing its tasks
Syslog severity level = 4

**WARNING_TRACE_NAME**

```java
public static final java.lang.String WARNING_TRACE_NAME
```

String descriptor for WARNING trace level

### Methods

**getName()**

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

Returns the name of this Trace object.

Returns: the name of this Trace object

**getSubFacility()**

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

Returns the subFacility of trace

Returns: the trace subFacility type
**getType()**

```java
public int getType()
```

Returns the type of trace.

**Returns:** the trace severity as specified in Syslog. DEBUGGING, INFORMATIONAL, WARNING, etc.

**isEnabled()**

```java
public boolean isEnabled()
```

Returns the state of this Trace object. By default, Trace objects are enabled, that is, println() method will always trace. The state may not be changed through this interface, however, this object may implement additional interfaces that allow the state to be changed.

**Returns:** true if tracing is enabled, false otherwise

**See Also:** ConditionalTrace

**println(Object)**

```java
public void println(java.lang.Object object)
```

Prints the string returned by the Object.toString() method and terminates the line as defined by the system.

**Parameters:**

- `object` - the object to be printed

**println(String)**

```java
public void println(java.lang.String message)
```

Prints a message in the same format as Trace.print() and terminates the line as defined by the system.

**Parameters:**

- `message` - the message to be printed

**println(String, Object)**

```java
public void println(java.lang.String mnemonic,
                    java.lang.Object object)
```
TraceManager

Declaration

public interface TraceManager

Description

The TraceManager interface defines the methods that allow applications trace management.

Prints the string returned by the Object.toString() method and terminates the line as defined by the system.

Parameters:
object - the object to be printed
mnemonic - the mnemonic mapped to message to be printed

println(String, String)

public void println(java.lang.String mnemonic,
java.lang.String message)

Prints a message in the same format as Trace.print() and terminates the line as defined by the system.

Parameters:
message - the message to be printed
mnemonic - the mnemonic mapped to message to be printed

setDefaultMnemonic(String)

public void setDefaultMnemonic(java.lang.String mnemonic)

Sets a default mnemonic for all messages printed out to this trace.

Parameters:
mnemonic - a mnemonic string
Typically, an application obtains only one TraceManager object. All Trace objects are created by default: Predefined Trace in accordance with Syslog definitions are:

ContitionalTraces: INFORMATIONAL, DEBUGGING, NOTIFICATION, WARNING
UnconditionalTraces: ERROR, CRITICAL, ALERTS, EMERGENCIES

Facilities/Sub-Facilities:

**Facility**: is a code consisting of two or more uppercase letters that indicate the facility to which the message refers. A facility can be a hardware device, a protocol, or a module of the system software.

**SubFacility**: is a code consisting of two or more uppercase letters that indicate the sub-facility to which the message refers. A sub-facility can be a hardware device component, a protocol unit, or a sub-module of the system software.

By default all 8 Conditional and UnConditional Traces are created for the Facility and 8 for each of the subFacilities In order to use the DEBUGGING trace for the parent FACILITY, for example, the application needs to use the getConditionalTrace( “DEBUGGING” ) method of this object.

In order to use the DEBUGGING trace for the SUBFACILITY, for example, the application needs to use the getConditionalTrace( SUBFACILITY + “.” + “DEBUGGING” ) method of this object or use the getConditionalTrace( SUBFACILITY , “DEBUGGING” ) method.

System wide TraceWriterManager is set through the setTraceWriterManager method provided by this interface.

The Trace Manager object also allows the application to enable or disable tracing for all trace through the enableAll() and disableAll() methods.

### Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addSubFacilities(java.lang.String names)</td>
</tr>
<tr>
<td>Sets a set of subFacilities for this TraceManager/Facility.</td>
</tr>
</tbody>
</table>
### Member Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addSubFacility(java.lang.String name)</td>
<td>Adds a single subFacility for this TraceManager/Facility.</td>
</tr>
<tr>
<td>void disableAll()</td>
<td>Disables tracing for all Trace objects managed by this TraceManager.</td>
</tr>
<tr>
<td>void disableTimeStamp()</td>
<td>Disables prefixing a time stamp for every message printed by this TraceManager.</td>
</tr>
<tr>
<td>void enableAll()</td>
<td>Enables tracing for all Trace objects managed by this TraceManager.</td>
</tr>
<tr>
<td>void enableTimeStamp()</td>
<td>Enables prefixing a time stamp for every message printed by this TraceManager.</td>
</tr>
<tr>
<td>ConditionalTrace getConditionalTrace(int severity)</td>
<td>Creates a new ConditionalTrace object or obtains an existing ConditionalTrace object for this condition.</td>
</tr>
<tr>
<td>ConditionalTrace getConditionalTrace(java.lang.String subFacility, int severity)</td>
<td>Creates a new ConditionalTrace object or obtains an existing ConditionalTrace object for this condition and subFacility.</td>
</tr>
<tr>
<td>java.lang.String getName()</td>
<td>Returns the Facility name for this TraceManager.</td>
</tr>
<tr>
<td>java.lang.String[] getSubFacilities()</td>
<td>Returns the subFacility names for this TraceManager/Facility.</td>
</tr>
<tr>
<td>java.util.Enumeration getTraces()</td>
<td>Returns an enumeration of the Trace objects managed by this TraceManager.</td>
</tr>
<tr>
<td>TraceWriterManager getTraceWriterManager()</td>
<td>Returns the TraceWriter used by this TraceManager.</td>
</tr>
<tr>
<td>UnconditionalTrace getUnconditionalTrace(int severity)</td>
<td>Creates a new UnconditionalTrace object or obtains an existing UnconditionalTrace object for this condition.</td>
</tr>
<tr>
<td>UnconditionalTrace getUnconditionalTrace(java.lang.String subFacility, int severity)</td>
<td>Creates a new UnconditionalTrace object or obtains an existing UnconditionalTrace object for this condition and subFacility.</td>
</tr>
<tr>
<td>void removeTrace(Trace tc)</td>
<td>Removes a Trace object given an object.</td>
</tr>
<tr>
<td>void setSubFacilities(java.lang.String names)</td>
<td>Sets a set of subFacilities for this TraceManager/Facility.</td>
</tr>
<tr>
<td>void setSubFacility(java.lang.String name)</td>
<td>Adds a single subFacility for this TraceManager/Facility.</td>
</tr>
<tr>
<td>void setTraceWriterManager(TraceWriterManager twm)</td>
<td>Sets the TraceWriter to be used by this TraceManager.</td>
</tr>
</tbody>
</table>
Methods

addSubFacilities(String[])

public void addSubFacilities(java.lang.String[] names)
Sets a set of subFacilities for this TraceManager/Facility.

addSubFacility(String)

public void addSubFacility(java.lang.String name)
Adds a single subFacility for this TraceManager/Facility.

disableAll()

public void disableAll()
Disables tracing for all Trace objects managed by this TraceManager.

disableTimeStamp()

public void disableTimeStamp()
Disables prefixing a time stamp for every message printed by this TraceManager.

enableAll()

public void enableAll()
Enables tracing for all Trace objects managed by this TraceManager.

enableTimeStamp()

public void enableTimeStamp()
Enables prefixing a time stamp for every message printed by this TraceManager.

getConditionalTrace(int)

public com.cisco.services.tracing.ConditionalTrace getConditionalTrace(int severity)
Creates a new ConditionalTrace object or obtains an existing ConditionalTrace object for this condition.
getConditionalTrace(String, int)

public com.cisco.services.tracing.ConditionalTrace
getConditionalTrace(java.lang.String subFacility,
int severity)

Creates a new ConditionalTrace object or obtains an existing
ConditionalTrace object for this condition and subFacility

getName()

public java.lang.String getName()

Returns the Facility name for this TraceManager.

getSubFacilities()

public java.lang.String[] getSubFacilities()

Returns the subFacility names for this TraceManager/Facility.

getTraces()

public java.util.Enumeration getTraces()

Returns an enumeration of the Trace objects managed by this TraceManager.

getTraceWriterManager()

public com.cisco.services.tracing.TraceWriterManager
getTraceWriterManager()

Returns the TraceWriter used by this TraceManager.

getUnconditionalTrace(int)

public com.cisco.services.tracing.UnconditionalTrace
getUnconditionalTrace(int severity)

Creates a new UnconditionalTrace object or obtains an existing
UnconditionalTrace object for this condition.

getUnconditionalTrace(String, int)

public com.cisco.services.tracing.UnconditionalTrace
getUnconditionalTrace(java.lang.String
subFacility, int severity)
Chapter 2 Cisco JTAPI Implementation

TraceManagerFactory

Creates a new UnconditionalTrace object or obtains an existing UnconditionalTrace object for this condition and subFacility

removeTrace(Trace)

public void removeTrace(com.cisco.services.tracing.Trace tc)

Removes a Trace object given an object.

setSubFacilities(String[])

public void setSubFacilities(java.lang.String[] names)

Deprecated. and replaced with TraceManager.addSubFacilities method

Sets a set of subFacilities for this TraceManager/Facility.

setSubFacility(String)

public void setSubFacility(java.lang.String name)

Deprecated. and replaced with TraceManager.addSubFacility method

Adds a single subFacility for this TraceManager/Facility.

setTraceWriterManager(TraceWriterManager)

public void setTraceWriterManager(com.cisco.services.tracing.TraceWriterManager twm)

Sets the TraceWriter to be used by this TraceManager.

TraceManagerFactory

Declaration

public class TraceManagerFactory

| java.lang.Object
| +-- com.cisco.services.tracing.TraceManagerFactory
The `TraceManagerFactory` class is a class by which applications obtain a TraceManager object. The TraceModule passed in the constructor is registered in a list. The list can be enumerated using the `getModules()` method.

### Methods

**getModules()**

```java
public static java.util.Enumeration getModules()
```

Returns an enumeration of the TraceModules registered with this factory.

### Inherited Member Summary

Methods inherited from class `Object`
- clone()
- equals(Object)
- finalize()
- getClass()
- hashCode()
- notify()
- notifyAll()
- toString()
- wait()
- wait()
- wait()
public static com.cisco.services.tracing.TraceManager
registerModule(TraceModule module)

Returns an instance of a TraceManager object. The contained
TraceWriterManager will not have any default TraceWriters.

public static com.cisco.services.tracing.TraceManager
registerModule(TraceModule, String[], TraceWriterManager
module, java.lang.String[] subFacilities,
com.cisco.services.tracing.TraceWriterManager trace
WriterManager)

Returns an instance of a TraceManager object. Trace output will be
redirected to the TraceWriterManager object specified.

public static com.cisco.services.tracing.TraceManager
registerModule(TraceModule, TraceWriterManager
module, com.cisco.services.tracing.TraceWriterManager trace
WriterManager)

Returns an instance of a TraceManager object. Trace output will be
redirected to the TraceWriterManager object specified.

public interface TraceModule

All Known Subinterfaces

com.cisco.jtapi.extensions.CiscoJtapiPeer
**Description**

The `TraceModule` interface serves two purposes. First, it allows applications to discover the TraceManager object used by other packages that they use. Second, applications that register with the TraceManagerFactory must identify themselves by implementing this interface.

### Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getTraceManager()</code></td>
<td>Returns the TraceManager that an object is using for tracing.</td>
</tr>
<tr>
<td><code>getTraceModuleName()</code></td>
<td>Returns the module name.</td>
</tr>
</tbody>
</table>

### Methods

**getTraceManager()**

```java
public com.cisco.services.tracing.TraceManager getTraceManager()
```

Returns the TraceManager that an object is using for tracing.

**getTraceModuleName()**

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

Returns the module name.

### TraceWriter Declaration

```java
public interface TraceWriter
```
All Known Subinterfaces

TraceWriterManager

All Known Implementing Classes

BaseTraceWriter

Description

The TraceWriter interface abstracts the details of trace message output. The TraceWriter uses its enabled method to advertise whether or not the print and println methods will have any effect. Users of TraceWriter should use the value returned by the getEnabled method as an indication of whether they should invoke the print and println methods at all.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
</table>
| void close()  
  Releases any resources associated by this TraceWriter. |
| void flush()  
  Forces output of any messages that have been printed using the println method |
| java.lang.String getDescription() |
| boolean getEnabled()  
  Returns whether the println method will print anything or not. |
| java.lang.String getName() |
| int[] getTraceLevels() |
| void println(java.lang.String message, int severity)  
  Prints the specified string followed by a carriage return The concrete TraceWriter class will use the severity to block out messages from a particular stream. |
| void setTraceLevels(int[] levels)  
  set the trace levels that will be traced by this TraceWriter |
Methods

close()

public void close()

Releases any resources associated by this TraceWriter.

flush()

public void flush()

Forces output of any messages that have been printed using the println method.

description()

public java.lang.String getDescription()

Returns: a short description of this TraceWriter

gEnabled()

public boolean getEnabled()

Returns whether the println method will print anything or not. A closed TraceWriter will always return false from this method.

Returns: true if this TraceWriter is enabled, false if not

getName()

public java.lang.String getName()

Returns: the name of this TraceWriter

getTraceLevels()

public int[] getTraceLevels()

Returns: the array of trace levels that will be traced by this TraceWriter

println(String, int)

public void println(java.lang.String message, int severity)

Prints the specified string followed by a carriage return. The concrete TraceWriter class will use the severity to block out messages from a
particular stream. Each trace writer has a notion of the highest level trace it traces

**Parameters:**
- message - the string to print
- severity - of the trace.

**See Also:**
- Trace

### setTraceLevels(int[])

```java
public void setTraceLevels(int[] levels)
```

set the trace levels that will be traced by this TraceWriter

**Parameters:**
- int[] - levels

**See Also:**
- Trace

---

**TraceWriterManager**

**Declaration**

```java
public interface TraceWriterManager extends TraceWriter
```

**All Superinterfaces**

```java
TraceWriter
```
Description

TraceWriterManager contains the list of TraceWriter objects that are used to implement the tracing. The list is populated at startup from the switches in a .ini file. A LogFileTraceWriter, a ConsoleTraceWriter, and a SyslogTraceWriter are available. Users can override the existing TraceWriters by setting a user implemented TraceWriter[] or adding to the existing TraceWriters. This makes it possible to add other TraceWriters that can function along with existing trace writers.

Member Summary

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addTraceWriter(TraceWriter traceWriter)</td>
<td>Add another TraceWriter to the array</td>
</tr>
<tr>
<td>TraceWriter[] getTraceWriters()</td>
<td>Remove the TraceWriter from the array in the manager</td>
</tr>
<tr>
<td>void removeTraceWriter(TraceWriter traceWriter)</td>
<td>Implementations can use this method to override or enhance the provided TraceWriters</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from interface **TraceWriter**

close(), flush(), getDescription(), getEnabled(), getName(), getTraceLevels(), println(String, int), setTraceLevels(int[])
### Methods

**addTraceWriter(TraceWriter)**

```
public void addTraceWriter(com.cisco.services.tracing.TraceWriter traceWriter)
```

Add another TraceWriter to the array

**Parameters:**
- TraceWriter - to be added to the list

**getTraceWriters()**

```
public com.cisco.services.tracing.TraceWriter[] getTraceWriters()
```

**Returns:** the array of TraceWriters in the manager

**removeTraceWriter(TraceWriter)**

```
public void removeTraceWriter(com.cisco.services.tracing.TraceWriter traceWriter)
```

Remove the TraceWriter from the array in the manager

**setTraceWriters(TraceWriter[])**

```
public void setTraceWriters(com.cisco.services.tracing.TraceWriter[] traceWriters)
```

Implementations can use this method to override or enhance the provided TraceWriters

**Parameters:**
- set - the array of TraceWriters.
UnconditionalTrace

Declaration

public interface UnconditionalTrace extends Trace

All Superinterfaces

Trace

Description

The UnconditionalTrace interface extends the Trace interface. Note that because this object extends Trace, its state is enabled by default and it may not be changed.

Typically, applications would obtain one UnconditionalTrace object per each condition that they need to trace always under any circumstances (such as, ERROR, FATAL, and so on).

Inherited Member Summary

<table>
<thead>
<tr>
<th>Fields inherited from interface Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALERTS, ALERTS_TRACE_NAME, CRITICAL, CRITICAL_TRACE_NAME, DEBUGGING, DEBUGGING_TRACE_NAME, EMERGENCIES, EMERGENCIES_TRACE_NAME, ERROR, ERROR_TRACE_NAME, HIGHEST_LEVEL, INFORMATIONAL, INFORMATIONAL_TRACE_NAME, LOWEST_LEVEL, NOTIFICATION, NOTIFICATION_TRACE_NAME, WARNING, WARNING_TRACE_NAME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods inherited from interface Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName(), getSubFacility(), getType(), isEnabled(), println(String, Object), println(String, Object), println(String, Object), println(String, Object), setDefaultMnemonic(String)</td>
</tr>
</tbody>
</table>
JTAPI Examples

This chapter provides the source code for makecall, the Cisco JTAPI program used to test the JTAPI installation. The makecall program is comprised of a series of programs written in Java using the Cisco JTAPI implementation.

This chapter contains the following sections:

- MakeCall.java
- Actor.java
- Originator.java
- Receiver.java
- StopSignal.java
- Trace.java
- TraceWindow.java

This chapter also provides instructions on how to invoke makecall:

- Running makecall
/**
 * makecall.java
 * 
 * Copyright Cisco Systems, Inc.
 * 
 * Performance-testing application (first pass) for Cisco JTAPI
 * implementation.
 * 
 * Known problems:
 * 
 * Due to synchronization problems between Actors, calls may
 * not be cleared when this application shuts down.
 * 
 */

//import com.ms.wfc.app.*;
import java.util.*;
import javax.telephony.*;
import javax.telephony.events.*;
import com.cisco.cti.util.Condition;

public class makecall extends TraceWindow implements ProviderObserver
{
    Vector actors = new Vector();
    Condition conditionInService = new Condition();
    Provider provider;

    public makecall ( String [] args ) {
        super ("makecall" + ": " + new CiscoJtapiVersion());
        try {

            println ( "Initializing Jtapi" );
            int curArg = 0;
            String providerName = args[curArg++];
            String login = args[curArg++];
            String passwd = args[curArg++];
            int actionDelayMillis = Integer.parseInt ( args[curArg++] );
            String src = null;
            String dest = null;

            JtapiPeer peer = JtapiPeerFactory.getJtapiPeer ( null );
            if ( curArg < args.length ) {

            }

        } catch (Exception e) {
            println ( "Exception in makecall: " + e.getMessage() );
            e.printStackTrace();

        }
    }
}
String providerString = providerName + ";login=" + login + ";passwd=" + passwd;
println ( "Opening " + providerString + "...\n" );
provider = peer.getProvider ( providerString );
provider.addObserver ( this );
conditionInService.waitTrue ();

println ( "Constructing actors" );

for (; curArg < args.length; curArg++) {
    if ( src == null ) {
        src = args[curArg];
    } else {
        dest = args[curArg];
        Originator originator = new Originator ( provider.getAddress ( src ), dest, this, actionDelayMillis );
        actors.addElement ( originator );
        actors.addElement ( new Receiver ( provider.getAddress ( dest ), this, actionDelayMillis, originator ) );
        src = null;
        dest = null;
    }
}
if ( src != null ) {
    println ( "Skipping last originating address " + src + "\n; no destination specified" );
}

Enumeration e = actors.elements ();
while ( e.hasMoreElements () ) { 
    Actor actor = (Actor) e.nextElement ();
    actor.initialize ();
}

Enumeration en = actors.elements ();
while ( en.hasMoreElements () ) { 
    Actor actor = (Actor) en.nextElement ();
    actor.start ();
}

} catch ( Exception e ) {
    println ( "Caught exception " + e );
}
public void dispose () {
    println ( "Stopping actors" );
    Enumeration e = actors.elements () ;
    while ( e.hasMoreElements () ) {
        Actor actor = (Actor) e.nextElement () ;
        actor.dispose ();
    }
}

public static void main ( String [] args )
{
    if ( args.length < 6 ) {
        System.out.println ( "Usage: makecall <server> <login> <password> <delay> <origin> <destination> ..." );
        System.exit ( 1 );
    }
    new makecall ( args );
}

public void providerChangedEvent ( ProvEv [] eventList ) {
    if ( eventList != null ) {
        for ( int i = 0; i < eventList.length; i++ )
        {
            if ( eventList[i] instanceof ProvInServiceEv ) {
                conditionInService.set ();
            }
        }
    }
}
public abstract class Actor implements AddressObserver, TerminalObserver, CallControlCallObserver, Trace {

public static final int ACTOR_OUT_OF_SERVICE = 0;
public static final int ACTOR_IN_SERVICE = 1;
private Trace trace;
protected int actionDelayMillis;
private Address observedAddress;
private Terminal observedTerminal;
private boolean addressInService;
private boolean terminalInService;
protected int state = Actor.ACTOR_OUT_OF_SERVICE;

public Actor ( Trace trace, Address observed, int actionDelayMillis ) {
    this.trace = trace;
    this.observedAddress = observed;
    this.observedTerminal = observed.getTerminals ()[0];
    this.actionDelayMillis = actionDelayMillis;
}

public void initialize () {
    try {
        if ( observedAddress != null ) {
            bufPrintln ( "Adding Call observer to address " + observedAddress.getName () );
            observedAddress.addCallObserver ( this );
        }
    //Now add observer on Address and Terminal
    }
}
bufPrintln {
    "Adding Address Observer to address "+ observedAddress.getName ()
    
};

observedAddress.addObserver ( this );

bufPrintln {
    "Adding Terminal Observer to Terminal"+ observedTerminal.getName ()
    
};

observedTerminal.addObserver ( this );

catch ( Exception e ) {
} finally {
    flush ();
}
}

public final void start () {
onStart ();
}

public final void dispose () {

try {
onStop ();
if ( observedAddress != null ) {
    bufPrintln {
        "Removing observer from Address "+ observedAddress.getName ()
    
};
    observedAddress.removeObserver ( this );

    bufPrintln {
        "Removing call observer from Address "+ observedAddress.getName ()
    
};
    observedAddress.removeCallObserver ( this );

} if ( observedTerminal != null ){
    bufPrintln {

}
"Removing observer from terminal "
+ observedTerminal.getName ()
);
observedTerminal.removeObserver ( this );
}
catch ( Exception e ) {
 println ( "Caught exception " + e );
}
finally {
 flush ();
}
}
public final void stop () {
 onStop ();
}
public final void callChangedEvent ( CallEv [] events ) {
 //
 // for now, all metaevents are delivered in the
 // same package...
 //
 metaEvent ( events );
}
public void addressChangedEvent ( AddrEv [] events ) {
 for ( int i=0; i<events.length; i++ ) {
 Address address = events[i].getAddress ();
 switch ( events[i].getID () ) {
 case CiscoAddrInServiceEv.ID:
 bufPrintln ( "Received " + events[i] + " for "+ address.getName ());
 addressInService = true;
 if ( terminalInService ) {
 if ( state != Actor.ACTOR_IN_SERVICE ) {
 state = Actor.ACTOR_IN_SERVICE ;
 fireStateChanged ();
 }
 break;
 case CiscoAddrOutOfServiceEv.ID:
 bufPrintln ( "Received " + events[i] + " for "+ address.getName ());
 addressInService = false;
 if ( state != Actor.ACTOR_OUT_OF_SERVICE ) {
 state = Actor.ACTOR_OUT_OF_SERVICE; // you only want to notify when
 you had notified earlier that you are IN_SERVICE
```java
fireStateChanged ();
} 
break;
}
flush ();
}

public void terminalChangedEvent ( TermEv [] events ) {
for ( int i=0; i<events.length; i++ ) {
    Terminal terminal = events[i].getTerminal ();
    switch ( events[i].getID () ) {
    case CiscoTermInServiceEv.ID:
        bufPrintln ( "Received " + events[i] + " for " + terminal.getName () );
        terminalInService = true;
        if ( addressInService ) {
            if ( state != Actor.ACTOR_IN_SERVICE ) {
                state = Actor.ACTOR_IN_SERVICE;
                fireStateChanged ();
            }
        }
        break;
    case CiscoTermOutOfServiceEv.ID:
        bufPrintln ( "Received " + events[i] + " for " + terminal.getName () );
        terminalInService = false;
        if ( state != Actor.ACTOR_OUT_OF_SERVICE ) { // you only want to notify when you had notified earlier that you are IN_SERVICE
            state = Actor.ACTOR_OUT_OF_SERVICE;
            fireStateChanged ();
        }
        break;
    }
    flush();
}

final void delay ( String action ) {
    if ( actionDelayMillis != 0 ) {
        println ( "Pausing " + actionDelayMillis + " milliseconds before " + action );
        try {
            Thread.sleep ( actionDelayMillis );
        } catch ( InterruptedException e ) {} 
    }
}

protected abstract void metaEvent ( CallEv [] events );
```
protected abstract void onStart ();
protected abstract void onStop ();
protected abstract void fireStateChanged ();

public final void bufPrint ( String string ) {
    trace.bufPrint ( string );
}

public final void bufPrintln ( String string ) {
    trace.bufPrint ( string );
    trace.bufPrint ("\n");
}

public final void print ( String string ) {
    trace.print ( string );
}

public final void print ( char character ) {
    trace.print ( character );
}

public final void print ( int integer ) {
    trace.print ( integer );
}

public final void println ( String string ) {
    trace.println ( string );
}

public final void println ( char character ) {
    trace.println ( character );
}

public final void println ( int integer ) {
    trace.println ( integer );
}

public final void flush () {
    trace.flush ();
}
import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcontrol.*;
import javax.telephony.callcontrol.events.*;
import com.ms.com.*;
import com.cisco.jtapi.extensions.*;

public class Originator extends Actor {
    public Originator ( Address srcAddress, String destAddress, Trace trace, int actionDelayMillis ) {
        super ( trace, srcAddress, actionDelayMillis );// observe srcAddress
        this.srcAddress = srcAddress;
        this.destAddress = destAddress;
        this.iteration = 0;
    }

    protected final void metaEvent ( CallEv [] eventList ) {
        for ( int i = 0; i < eventList.length; i++ ) {
            try {
                CallEv curEv = eventList[i];
                if ( curEv instanceof CallCtlTermConnTalkingEv ) {
                    TerminalConnection tc = ((CallCtlTermConnTalkingEv)curEv).getTerminalConnection ();
                    Connection conn = tc.getConnection ();
                    if ( conn.getAddress ().getName ().equals ( destAddress ) ) {
                        delay ( "disconnecting" );
                        bufPrintln ( "Disconnecting Connection " + conn );
                    }
                }
            } catch ( Exception e ) {
                
            }
        }
    }
}
Originator.java

```java
protected void makecall ()
    throws ResourceUnavailableException, InvalidStateException,
           PrivilegeViolationException, MethodNotSupportedException,
           InvalidPartyException, InvalidArgumentException {
    println ( "Making call #" + ++iteration + " from " + srcAddress + " to " +
              destAddress + " " + Thread.currentThread ().getName () );
    Call call = srcAddress.getProvider ().createCall ();
    call.connect ( srcAddress.getTerminals ()[0], srcAddress, destAddress );
    setCallProgressState ( false );
    println ( "Done making call" );
}

protected final void onStart () {
    stopSignal = new StopSignal ();
    new ActionThread ().start ();
}

protected final void fireStateChanged () {
    checkReadyState ();
}

protected final void onStop () {
    stopSignal.stop ();
    Connection[] connections = srcAddress.getConnections ();
    try {
        if ( connections != null ) {
```
for (int i=0; i< connections.length; i++) {
    connections[i].disconnect();
}

} catch ( Exception e ) {
    println (" Caught Exception " + e);
}

}

public int getReceiverState () {
    return receiverState;
}

public void setReceiverState ( int state ) {
    if ( receiverState != state ){
        receiverState = state;
        checkReadyState ();
    }
}

public synchronized void checkReadyState () {
    if ( receiverState == Actor.ACTOR_IN_SERVICE && state == Actor.ACTOR_IN_SERVICE )
        ready = true;
    else {
        ready = false;
    }
    notifyAll ();
}

public synchronized void setCallProgressState ( boolean isCallInIdle ) {
    callInIdle = isCallInIdle;
    notifyAll ();
}

public synchronized void doAction () {
    if ( !ready || !callInIdle ) {
        try {
            wait ();
        } catch ( Exception e ) {
            println (" Caught Exception from wait state" + e);
        }
    } else {
        if ( actionDelayMillis != 0 ) {
            println ("Pausing " + actionDelayMillis + " milliseconds before making call ");
            // perform the action
        } else {
            // do something else
        }
    }
}
flush ();
try {
    wait ( actionDelayMillis );
} catch ( Exception ex ) {} 

//make call after waking up, recheck the flags before making the call 
if ( ready && callInIdle ) {
    try {
        makecall ();
    } catch ( Exception e ) {
        println ( "Caught Exception in MakeCall " + e + " Thread =" + Thread.currentThread ().getName () );
    }
}
}

class ActionThread extends Thread {
    ActionThread ( ) {
        super ( "ActionThread" );
    }

    public void run () {
        while ( true ) {
            doAction ();
        }
    }
}

Receiver.java

/**
 * Receiver.java
 *
 * Copyright Cisco Systems, Inc.
 *
 */

import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcontrol.*;
import javax.telephony.callcontrol.events.*;

public class Receiver extends Actor {
    Address address;
    StopSignal stopSignal;
    Originator originator;

    public Receiver ( Address address, Trace trace, int actionDelayMillis, Originator originator ) {
        super ( trace, address, actionDelayMillis );
        this.address = address;
        this.originator = originator;
    }

    protected final void metaEvent ( CallEv [] eventList ) {
        for ( int i = 0; i < eventList.length; i++ ) {
            TerminalConnection tc = null;
            try {
                CallEv curEv = eventList[i];
                if ( curEv instanceof CallCtlTermConnRingingEv ) {
                    tc = (CallCtlTermConnRingingEv)curEv).getTerminalConnection ();
                    delay ( "answering" );
                    bufPrintln ( "Answering TerminalConnection " + tc);
                    tc.answer ();
                    stopSignal.canStop ();
                }
            } catch ( Exception e ) {
                bufPrintln ( "Caught exception " + e );
                bufPrintln ( "tc = " + tc );
            }
        } finally {
            flush ();
        }
    }
}
protected final void onStart () {
    stopSignal = new StopSignal ();
}

protected final void onStop () {
    stopSignal.stop ();
    Connection[] connections = address.getConnections ();
    try {
        if ( connections != null ) {
            for (int i=0; i< connections.length; i++ ) {
                connections[i].disconnect ();
            }
        }
    }catch ( Exception e ) {
        println (" Caught Exception " + e);
    }
}

protected final void fireStateChanged () {
    originator.setReceiverState ( state );
}
StopSignal.java

/**
 * StopSignal.java
 * 
 * Copyright Cisco Systems, Inc.
 * 
 */

class StopSignal {
    boolean stopping = false;
    boolean stopped = false;
    synchronized boolean isStopped () {
        return stopped;
    }
    synchronized boolean isStopping () {
        return stopping;
    }
    synchronized void stop () {
        if ( !stopped ) {
            stopping = true;
            try {
                wait ();
            } catch ( InterruptedException e ) {}
        }
    }
    synchronized void canStop () {
        if ( stopping = true ) {
            stopping = false;
            stopped = true;
            notify ();
        }
    }
}
public interface Trace
{
    /**
     * bufPrint (str) puts str in buffer only.
     */
    public void bufPrint ( String string );

    /**
     * print () println () bufPrint and invoke flush ()..
     */
    public void print ( String string );
    public void print ( char character );
    public void print ( int integer );
    public void println ( String string );
    public void println ( char character );
    public void println ( int integer );

    /**
     * flush out the buffer.
     */
    public void flush ();
}
TraceWindow.java

import java.awt.*;
import java.awt.event.*;

public class TraceWindow extends Frame implements Trace {

    TextArea textArea;
    boolean traceEnabled = true;
    StringBuffer buffer = new StringBuffer();

    public TraceWindow (String name) {
        super (name);
        initWindow();
    }

    public TraceWindow() {
        this("");
    }

    private void initWindow() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                dispose();
            }
        });
        textArea = new TextArea();
        setSize(400, 400);
        add(textArea);
        setEnabled(true);
        this.show();
    }

    public final void bufPrint (String str) {
        if (traceEnabled) {
            textArea.append(str);
            buffer.append(str);
        }
    }
}
public final void print ( String str ) {
    if ( traceEnabled ) {
        buffer.append ( str );
        flush ();
    }
}

public final void print ( char character ) {
    if ( traceEnabled ) {
        buffer.append ( character );
        flush ();
    }
}

public final void println ( String str ) {
    if ( traceEnabled ) {
        print ( str );
        print ( '\n' );
        flush ();
    }
}

public final void println ( char character ) {
    if ( traceEnabled ) {
        print ( character );
        print ( '\n' );
        flush ();
    }
}

public final void println ( int integer ) {
    if ( traceEnabled ) {
        print ( integer );
        print ( '\n' );
        flush ();
    }
}

public final void setTrace ( boolean traceEnabled ) {
    this.traceEnabled = traceEnabled;
}
public final void flush () {
    if (traceEnabled) {
        textArea.append(buffer.toString());
        buffer = new StringBuffer();
    }
}

public final void clear () {
    textArea.setText(""sv);
Running makecall

Invoking makecall

On the client workstation

From the Windows NT command line, navigate to the makecall directory where JTAPI Tools directory was installed and execute the following command:

```
jview makecall <server name> <login> <password> 1000 <device 1><device2>
```

<server name> is the hostname or IP address of your Cisco CallManager and <device1> <device2> are directory numbers of IP phones. The phones must be part of the associated devices of a given user as administered in the Cisco CallManager’s directory administration web page. The <logic> and <password> are similarly as administered in the directory. This will test that you have installed and configured everything correctly. The application will make calls between the two devices with an action delay of 1000 msecs until terminated.
Running makecall
This appendix contains a listing of all the classes and interfaces available in the Cisco JTAPI implementation for Cisco CallManager:

- Cisco JTAPI Version 1.2 Classes and Interfaces, page A-2, which lists all the JTAPI v 1.2 classes and methods. The supported classes and methods have a check mark in the Cisco JTAPI Support column.

- Cisco JTAPI Extension Classes and Interfaces, page A-23, which lists the Cisco extension classes and methods.

- Cisco Trace Logging Classes and Interfaces, page A-27, which lists the error tracing classes and methods.
## Cisco J TAPI Version 1.2 Classes and Interfaces

### Core Package

Table A-1 lists each J TAPI interface in the J TAPI Core Package followed by the associated method(s) and whether the classes are supported by the Cisco J TAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>addCallObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addressObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAddressCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallObservers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConnections</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getObservers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProvider</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminals</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeCallObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>AddressObserver</td>
<td>addressChangedEvent</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td>addObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>connect</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>capabilities</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

A CallObserver must exist for the Terminal or Address originating the call.
### Table A-1: Support for `javax.telephony` (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getObservers</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getProvider</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getState</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeObserver</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallObserver</td>
<td>callChangedEvent</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Connection</td>
<td>disconnect</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>getCall</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getCapabilities</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getState</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getTerminalConnections</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JtapiPeer</td>
<td>getName</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>getProvider</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getServices</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JtapiPeerFactory</td>
<td>getJtapiPeer</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Provider</td>
<td>addObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>createCall</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAddress</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAddressCapabilities()</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAddressCapabilities(Terminal)</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAddresses</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getCallCapabilities()</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1  Support for javax.telephony (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getCallCapabilities(Terminal, Address)</td>
<td>Yes</td>
<td>This method returns calls only when there are CallObservers attached to Addresses or Terminals, when a RouteAddress is registered for routing, or when a CiscoMediaTerminal is registered.</td>
</tr>
<tr>
<td></td>
<td>getCalls</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConnectionCapabilities()</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConnectionCapabilities(Terminal, Address)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getObservers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProviderCapabilities()</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProviderCapabilities(Terminal)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminal</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalCapabilities()</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalCapabilities()</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalConnectionCapabilities()</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalConnectionCapabilities(Terminal)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminals</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeObserver</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1 Support for javax.telephony (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>shutdown</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ProviderObserver</td>
<td>providerChangedEvent</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Terminal</td>
<td>addCallObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAddresses</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallObservers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getObservers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProvider</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalConnections</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeCallObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeObserver</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TerminalConnection</td>
<td>answer</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConnection</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminal</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTerminalConnectionCapabilities</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>TerminalObserver</td>
<td>terminalChangedEvent</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Call Center Package

Table A-2 lists each JTAPI interface in the JTAPI Call Center Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDAddress</td>
<td>getACDManagerAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLoggedOnAgents</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getNumberQueued</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getOldestCallQueued</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getQueueWaitTime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getRelativeQueueLoad</td>
<td></td>
</tr>
<tr>
<td>ACDAddressObserver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDConnection</td>
<td>getACDManagerConnection</td>
<td></td>
</tr>
<tr>
<td>ACDManagerAddress</td>
<td>getACDAddresses</td>
<td></td>
</tr>
<tr>
<td>ACDManagerConnection</td>
<td>getACDCollections</td>
<td></td>
</tr>
<tr>
<td>Agent</td>
<td>getACDAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgentID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setState</td>
<td></td>
</tr>
<tr>
<td>AgentTerminal</td>
<td>addAgent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgents</td>
<td></td>
</tr>
<tr>
<td></td>
<td>removeAgents</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setAgents</td>
<td></td>
</tr>
<tr>
<td>AgentTerminalObserver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCenterAddress</td>
<td>addCallObserver</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-2  Support for `javax.telephony.callcenter` (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallCenterCall</td>
<td>connectPredictive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getApplicationData</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTrunks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setApplicationData</td>
<td></td>
</tr>
<tr>
<td>CallCenterCallObserver</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCenterProvider</td>
<td>getACDAddresses</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getACDManagerAddresses</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getRouteableAddresses</td>
<td></td>
</tr>
<tr>
<td>CallCenterTrunk</td>
<td>getCall</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getType</td>
<td></td>
</tr>
<tr>
<td>RouteAddress</td>
<td>cancelRouteCallback</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getActiveRouteSessions</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getRouteCallback</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>registerRouteCallback</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteCallback</td>
<td>reRouteEvent</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>routeCallbackEndedEvent</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>routeEndEvent</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>routeEvent</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>routeUsedEvent</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteSession</td>
<td>endRoute</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getCause</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getRouteAddress</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>selectRoute</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Call Center Capabilities Package

Table A-3 lists each JTAPI interface in the JTAPI Call Center Capabilities Package followed by the associated method(s), and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDAddressCapabilities</td>
<td>canGetACDManagerAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetLoggedOnAgents</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetNumberQueued</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetOldestCallQueued</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetQueueWaitTime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetRelativeQueueLoad</td>
<td></td>
</tr>
<tr>
<td>ACDConnectionCapabilities</td>
<td>canGetACDManagerConnection</td>
<td></td>
</tr>
<tr>
<td>ACDManagerAddressCapabilities</td>
<td>canGetACDAddresses</td>
<td></td>
</tr>
<tr>
<td>ACDManagerConnectionCapabilities</td>
<td>canGetACDConnections</td>
<td></td>
</tr>
<tr>
<td>AgentTerminalCapabilities</td>
<td>canHandleAgents</td>
<td></td>
</tr>
<tr>
<td>CallCenterAddressCapabilities</td>
<td>canAddCallObserver</td>
<td></td>
</tr>
<tr>
<td>CallCenterCallCapabilities</td>
<td>canConnectPredictive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canGetTrunks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>canHandleApplicationData</td>
<td></td>
</tr>
<tr>
<td>CallCenterProviderCapabilities</td>
<td>canGetACDAddresses</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGetACDManagerAddresses</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGetRouteableAddresses</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteAddressCapabilities</td>
<td>canRouteCalls</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Call Center Events Package

Table A-4 lists each JTAPI interface in the JTAPI Call Center Events Package followed by the associated method(s), and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACDAddrBusyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrEv</td>
<td>getAgent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgentAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgentTerminal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTrunks</td>
<td></td>
</tr>
<tr>
<td>ACDAddrLoggedOffEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrLoggedOnEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrNotReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrUnknownEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrWorkNotReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACDAddrWorkReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermBusyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermEv</td>
<td>getACDAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgentAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAgentID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getState</td>
<td></td>
</tr>
<tr>
<td>AgentTermLoggedOffEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermLoggedOnEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermNotReadyEv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Support for `javax.telephony.callcenter.events` (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentTermReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermUnknownEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermWorkNotReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AgentTermWorkReadyEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCentCallAppDataEv</td>
<td>getApplicationData</td>
<td></td>
</tr>
<tr>
<td>CallCentCallEv</td>
<td>getCalledAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallingAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallingTerminal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLastRedirectedAddress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTrunks</td>
<td></td>
</tr>
<tr>
<td>CallCentConnEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCentConnInProgressEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCentEv</td>
<td>getCallCenterCause</td>
<td></td>
</tr>
<tr>
<td>CallCentTrunkEv</td>
<td>getTrunk</td>
<td></td>
</tr>
<tr>
<td>CallCentTrunkInvalidEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCentTrunkValidEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReRouteEvent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RouteCallbackEndedEvent</td>
<td>getRouteAddress</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteEndEvent</td>
<td>getCallingAddress</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getCallingTerminal</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getSetupInformation</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteSessionEvent</td>
<td>getRouteSession</td>
<td>Yes</td>
</tr>
<tr>
<td>RouteUsedEvent</td>
<td>getCallingAddress</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getCallingTerminal</td>
<td>Yes</td>
</tr>
<tr>
<td>Class Names</td>
<td>Method Names</td>
<td>Cisco JTAPI Support</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>getDomain</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getRouteUsed</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Call Control Package

Table A-5 lists each JTAPI interface in the JTAPI Call Control Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI Implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallControlAddress</td>
<td>cancelForwarding</td>
<td>Yes</td>
<td>Only for Call Forward All</td>
</tr>
<tr>
<td></td>
<td>getDoNotDisturb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getForwarding</td>
<td>Yes</td>
<td>Only for Call Forward All</td>
</tr>
<tr>
<td></td>
<td>getMessageWaiting</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>setDoNotDisturb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>setForwarding</td>
<td>Yes</td>
<td>Only for Call Forward All</td>
</tr>
<tr>
<td></td>
<td>setMessageWaiting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallControlCall</td>
<td>addParty</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>conference</td>
<td>Yes</td>
<td>In a consult conference scenario, only OriginalCall.conference (ConsultCall) is supported. ConsultCall.conference (OriginalCall) is not supported.</td>
</tr>
<tr>
<td></td>
<td>consult(TerminalConnection)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>consult(TerminalConnection, String)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>drop</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCalledAddress</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-5  Support for javax.telephony.callcontrol (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getCallingAddress</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallingTerminal</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConferenceController</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getConferenceEnable</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLastRedirectedAddress</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTransferController</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTransferEnable</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>offHook</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setConferenceController</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setConferenceEnable</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setTransferController</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setTransferEnable</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transfer(Call)</td>
<td>Yes</td>
<td>In a consult transfer scenario, only OriginalCall.transfer (ConsultCall) is supported. ConsultCall.transfer (OriginalCall) is not supported.</td>
</tr>
<tr>
<td></td>
<td>transfer(String)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CallControlCallObserver</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CallControlConnection</td>
<td>accept</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addToAddress</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCallControlState</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>park</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-5 Support for javax.telephony.callcontrol (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>redirect</td>
<td>Yes</td>
<td>Redirect allows a connection in the CallControlConnection. ESTABLISHED state to be redirected.</td>
</tr>
<tr>
<td></td>
<td>reject</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CallControlForwarding</td>
<td>getDestinationAddress</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getFilter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSpecificCaller</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>getType</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallControlTerminal</td>
<td>getDoNotDisturb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickup (Address, Address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickup (Connection, Address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickup (TerminalConnection, Address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickupFromGroup(Address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickupFromGroup(String, Address)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>setDoNotDisturb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallControlTerminalConnection</td>
<td>getCallControlState</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hold</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>join</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>leave</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unhold</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CallControlTerminalObserver</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Call Control Capabilities Package

Table A-6 lists each JTAPI interface in the JTAPI Call Control Capabilities Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallControlAddressCapabilities</td>
<td>canCancelForwarding</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGetDoNotDisturb</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGetForwarding</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGetMessageWaiting</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetDoNotDisturb</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetForwarding</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetMessageWaiting</td>
<td>Yes</td>
</tr>
<tr>
<td>CallControlCallCapabilities</td>
<td>canAddParty</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canConference</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canConsult</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canConsult(TerminalConnection)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canConsult(TerminalConnection, String)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canDrop</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canOffHook</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetConferenceController</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetConferenceEnable</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetTransferController</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetTransferEnable</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canTransfer</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canTransfer(Call)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Call Control Events Package

Table A-7 lists each JTAPI interface in the JTAPI Call Control Events Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallControlConnectionCapabilities</td>
<td>canTransfer(String)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canAccept</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canAddToAddress</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPark</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canRedirect</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canReject</td>
<td>Yes</td>
</tr>
<tr>
<td>CallControlTerminalCapabilities</td>
<td>canGetDoNotDisturb</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickup</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickup(Address, Address)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickup(Connection, Address)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickup(TerminalConnection, Address)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickupFromGroup</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickupFromGroup(_address)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canPickupFromGroup(String, Address)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canSetDoNotDisturb</td>
<td>Yes</td>
</tr>
<tr>
<td>CallControlTerminalConnectionCapabilities</td>
<td>canHold</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canJoin</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canLeave</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canUnhold</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Table A-7 Support for javax.telephony.callcontrol.events

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallCtlAddrDoNotDisturbEv</td>
<td>getDoNotDisturbState</td>
<td></td>
</tr>
<tr>
<td>CallCtlAddrEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCtlAddrForwardEv</td>
<td>getForwarding</td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlAddrMessageWaitingEv</td>
<td>getMessageWaitingState</td>
<td></td>
</tr>
<tr>
<td>CallCtlCallEv</td>
<td>getCalledState</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getCallingAddress</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getCallingTerminal</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getLastRedirectedAddress</td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnAlertingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnDialingEv</td>
<td>getDigits</td>
<td></td>
</tr>
<tr>
<td>CallCtlConnDisconnectedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnEstablishedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnInitiatedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnNetworkAlertingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnNetworkReachedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnOfferedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnQueuedEv</td>
<td>getNumberInQueue</td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlConnUnknownEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlEv</td>
<td>getCallControlCause</td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnBridgedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnDroppedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnHeldEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnInUseEv</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table A-7  Support for javax.telephony.callcontrol.events (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallCtlTermConnRingingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnTalkingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermConnUnknownEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallCtlTermDoNotDisturbEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallCtlTermEv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Capabilities Package

Table A-8  Support for javax.telephony.capabilities

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco J TAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressCapabilities</td>
<td>isObservable</td>
<td>Yes</td>
</tr>
<tr>
<td>CallCapabilities</td>
<td>canConnect</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>isObservable</td>
<td>Yes</td>
</tr>
<tr>
<td>ConnectionCapabilities</td>
<td>canDisconnect</td>
<td>Yes</td>
</tr>
<tr>
<td>ProviderCapabilities</td>
<td>isObservable</td>
<td>Yes</td>
</tr>
<tr>
<td>TerminalCapabilities</td>
<td>isObservable</td>
<td>Yes</td>
</tr>
<tr>
<td>TerminalConnectionCapabilities</td>
<td>canAnswer</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Events Package

Table A-9 lists each JTAPI interface in the JTAPI Events Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI Implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddrEv</td>
<td>getAddress</td>
<td>Yes</td>
</tr>
<tr>
<td>AddrObservationEndedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallActiveEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallEv</td>
<td>getCall</td>
<td>Yes</td>
</tr>
<tr>
<td>CallInvalidEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>CallObservationEndedEv</td>
<td>getEndedObject</td>
<td>Yes</td>
</tr>
<tr>
<td>ConnAlertingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnConnectedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnCreatedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnDisconnectedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnEv</td>
<td>getConnection</td>
<td>Yes</td>
</tr>
<tr>
<td>ConnFailedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnInProgressEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ConnUnknownEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Ev</td>
<td>getCaused</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getID</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getObserved</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>isNewMetaEvent</td>
<td>Yes</td>
</tr>
<tr>
<td>ProvEv</td>
<td>getProvider</td>
<td>Yes</td>
</tr>
<tr>
<td>ProvInServiceEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ProvObservationEndedEv</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
Media Package

Table A-9 lists each JTAPI interface from the JTAPI Media Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

Table A-10 Support for javax.telephony.events (continued)

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProvOutOfServiceEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>ProvShutdownEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnActiveEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnCreatedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnDroppedEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnEvgetTerminalConnection</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnPassiveEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TermConnRingingEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermConnUnknownEv</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>TermEv</td>
<td>getTerminal</td>
<td>Yes</td>
</tr>
<tr>
<td>TermObservationEndedEv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A-10 Support for javax.telephony.media

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>MediaCallObserver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MediaTerminalConnection</td>
<td>generateDtmf</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>getMediaAvailability</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setDtmfDetection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>startPlaying</td>
<td></td>
</tr>
<tr>
<td></td>
<td>startRecording</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stopPlaying</td>
<td></td>
</tr>
</tbody>
</table>
### Media Capabilities Package

Table A-10 lists each JTAPI interface in the JTAPI Media Capabilities Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

Table A-11 Support for javax.telephony.media.capabilities

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>MediaTerminalConnectionCapabilities</td>
<td>canDetectDtmf</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canGenerateDtmf</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canStartPlaying</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canStartRecording</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canStopPlaying</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canStopRecording</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canUseDefaultMicrophone</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canUseDefaultSpeaker</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canUsePlayURL</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>canUseRecordURL</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Media Events Package

Table A-12 lists each JTAPI interface in the JTAPI Media Events Package followed by the associated method(s) and whether the classes are supported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Class Names</th>
<th>Method Names</th>
<th>Cisco JTAPI Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>MediaEv</td>
<td>getMediaCause</td>
<td>Yes</td>
</tr>
<tr>
<td>MediaTermConnAvailableEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MediaTermConnDtmfEv</td>
<td>getDtmfDigit</td>
<td>Yes</td>
</tr>
<tr>
<td>MediaTermConnEv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MediaTermConnStateEv</td>
<td>getMediaState</td>
<td></td>
</tr>
<tr>
<td>MediaTermConnUnavailableEv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsupported Packages

Table A-13 shows the JTAPI packages that are unsupported by the Cisco JTAPI implementation.

<table>
<thead>
<tr>
<th>Unsupported JTAPI Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAPI Phone Package</td>
</tr>
<tr>
<td>JTAPI Phone Capabilities Package</td>
</tr>
<tr>
<td>JTAPI Phone Events Package</td>
</tr>
<tr>
<td>JTAPI Private Data Package</td>
</tr>
<tr>
<td>JTAPI Private Data Capabilities Package</td>
</tr>
<tr>
<td>JTAPI Private Data Events Package</td>
</tr>
</tbody>
</table>
Cisco J TAPI Extension Classes and Interfaces

Cisco J TAPI Extension Classes

Table A-14  Cisco J TAPI Extension Classes

<table>
<thead>
<tr>
<th>Cisco Extension Classes</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoMediaCapability</td>
<td>getMaxFramesPerPacket()</td>
</tr>
<tr>
<td></td>
<td>getPayloadType()</td>
</tr>
<tr>
<td></td>
<td>toString()</td>
</tr>
<tr>
<td>CiscoG711MediaCapability</td>
<td>getBitRate()</td>
</tr>
<tr>
<td></td>
<td>toString()</td>
</tr>
<tr>
<td>CiscoG723MediaCapability</td>
<td></td>
</tr>
<tr>
<td>RegistrationException</td>
<td></td>
</tr>
<tr>
<td>UnregistrationException</td>
<td></td>
</tr>
</tbody>
</table>

Cisco J TAPI Extension Interfaces

Table A-15  Cisco J TAPI Extension Interfaces and Their Methods

<table>
<thead>
<tr>
<th>Cisco Extension Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoAddrCreatedEv</td>
<td>getAddress()</td>
</tr>
<tr>
<td>CiscoAddress</td>
<td>getType()</td>
</tr>
<tr>
<td>CiscoAddressObserver</td>
<td></td>
</tr>
<tr>
<td>CiscoAddrEv</td>
<td></td>
</tr>
<tr>
<td>CiscoAddrOutOfService</td>
<td></td>
</tr>
<tr>
<td>CiscoCall</td>
<td>getCallID()</td>
</tr>
</tbody>
</table>
### Table A-15 Cisco J TAPI Extension Interfaces and Their Methods (continued)

<table>
<thead>
<tr>
<th>Cisco Extension Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoCallEv</td>
<td></td>
</tr>
<tr>
<td>CiscoCallID</td>
<td>getCall()</td>
</tr>
<tr>
<td></td>
<td>intValue()</td>
</tr>
<tr>
<td>CiscoConferenceEndEv</td>
<td>getConferenceCall()</td>
</tr>
<tr>
<td></td>
<td>getFinalCall()</td>
</tr>
<tr>
<td></td>
<td>getHeldConferenceController()</td>
</tr>
<tr>
<td></td>
<td>getTalkingConferenceController()</td>
</tr>
<tr>
<td>CiscoConferenceStartEv</td>
<td>getConferenceCall()</td>
</tr>
<tr>
<td></td>
<td>getFinalCall()</td>
</tr>
<tr>
<td></td>
<td>getHeldConferenceController()</td>
</tr>
<tr>
<td></td>
<td>getTalkingConferenceController()</td>
</tr>
<tr>
<td>CiscoConnection</td>
<td>getConnectionID()</td>
</tr>
<tr>
<td></td>
<td>getReason()</td>
</tr>
<tr>
<td>CiscoConnectionID</td>
<td>getConnection()</td>
</tr>
<tr>
<td></td>
<td>intValue()</td>
</tr>
<tr>
<td>CiscoConsultCall</td>
<td>getConsultingTerminalConnection()</td>
</tr>
<tr>
<td>CiscoConsultCallActiveEv</td>
<td>getHeldTerminalConnection()</td>
</tr>
<tr>
<td>CiscoEv</td>
<td></td>
</tr>
<tr>
<td>CiscoJtapiPeer</td>
<td></td>
</tr>
<tr>
<td>CiscoMediaTerminal</td>
<td>getRTPInputProperties()</td>
</tr>
<tr>
<td></td>
<td>getRTPOutputProperties()</td>
</tr>
<tr>
<td></td>
<td>register(InetAddress, int)</td>
</tr>
<tr>
<td></td>
<td>unregister()</td>
</tr>
<tr>
<td>CiscoProvEv</td>
<td></td>
</tr>
<tr>
<td>Cisco Extension Interfaces</td>
<td>Method Names</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>CiscoProvider</td>
<td>getCallbackGuardEnabled()</td>
</tr>
<tr>
<td></td>
<td>getMediaTerminal()</td>
</tr>
<tr>
<td></td>
<td>getMediaTerminals()</td>
</tr>
<tr>
<td></td>
<td>setCallbackGuardEnabled()</td>
</tr>
<tr>
<td>CiscoProviderObserver</td>
<td>getCall()</td>
</tr>
<tr>
<td>CiscoRouteSession</td>
<td>getRTPInputProperties()</td>
</tr>
<tr>
<td>CiscoRTPInputProperties</td>
<td>getBitRate()</td>
</tr>
<tr>
<td></td>
<td>getEchoCancellation()</td>
</tr>
<tr>
<td></td>
<td>getLocalAddress()</td>
</tr>
<tr>
<td></td>
<td>getLocalPort()</td>
</tr>
<tr>
<td></td>
<td>getPacketSize()</td>
</tr>
<tr>
<td></td>
<td>getPayloadType()</td>
</tr>
<tr>
<td>CiscoRTPInputStartedEv</td>
<td>getRTPInputProperties()</td>
</tr>
<tr>
<td>CiscoRTPInputStoppedEv</td>
<td>getBitRate()</td>
</tr>
<tr>
<td></td>
<td>getMaxFramesPerPacket()</td>
</tr>
<tr>
<td></td>
<td>getPacketSize()</td>
</tr>
<tr>
<td></td>
<td>getPayloadType()</td>
</tr>
<tr>
<td></td>
<td>getPrecedenceValue()</td>
</tr>
<tr>
<td></td>
<td>getRemoteAddress()</td>
</tr>
<tr>
<td></td>
<td>getRemotePort()</td>
</tr>
<tr>
<td>CiscoRTPOutputProperties</td>
<td>getRTPOutputProperties()</td>
</tr>
<tr>
<td>CiscoRTPOutputStoppedEv</td>
<td>getTerminal()</td>
</tr>
<tr>
<td>CiscoSynchronousObserver</td>
<td>getTerminal()</td>
</tr>
<tr>
<td>CiscoTermCreatedEv</td>
<td>getTerminal()</td>
</tr>
<tr>
<td>CiscoTermEv</td>
<td>getTerminal()</td>
</tr>
</tbody>
</table>
### Table A-15  Cisco JTAPI Extension Interfaces and Their Methods (continued)

<table>
<thead>
<tr>
<th>Cisco Extension Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiscoTerminal</td>
<td>getRegistrationState()</td>
</tr>
<tr>
<td>CiscoTerminalConnection</td>
<td></td>
</tr>
<tr>
<td>CiscoTerminalObserver</td>
<td></td>
</tr>
<tr>
<td>CiscoTermInServiceEv</td>
<td></td>
</tr>
<tr>
<td>CiscoTermOutOfServiceEv</td>
<td></td>
</tr>
<tr>
<td>CiscoTransferEndEv</td>
<td>getFinalCall()</td>
</tr>
<tr>
<td></td>
<td>getTransferController()</td>
</tr>
<tr>
<td></td>
<td>getTransferredCall()</td>
</tr>
<tr>
<td>CiscoTransferStartEv</td>
<td>getFinalCall()</td>
</tr>
<tr>
<td></td>
<td>getTransferController()</td>
</tr>
<tr>
<td></td>
<td>getTransferredCall()</td>
</tr>
<tr>
<td>ObjectContainer</td>
<td>getObject()</td>
</tr>
<tr>
<td></td>
<td>setObject()</td>
</tr>
<tr>
<td>RTPBitRate</td>
<td></td>
</tr>
<tr>
<td>RTPPayload</td>
<td></td>
</tr>
</tbody>
</table>
## Cisco Trace Logging Classes and Interfaces

### Cisco Trace Logging Classes

<table>
<thead>
<tr>
<th>Cisco Trace Logging Class</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogFileOutputStream</td>
<td>close(), flush(), getCurrentFile(), getFileExtension(), getFileNameBase(), getMaxFiles(), getMaxFileSize(), write(byte[], int, int), write(int)</td>
</tr>
<tr>
<td>NullTraceWriter</td>
<td>close(), flush(), getEnabled(), print(String), println(String)</td>
</tr>
</tbody>
</table>
## Table A-16 Cisco Trace Logging Classes

<table>
<thead>
<tr>
<th>Cisco Trace Logging Class</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputStreamTraceWriter</td>
<td>close()</td>
</tr>
<tr>
<td></td>
<td>flush()</td>
</tr>
<tr>
<td></td>
<td>getEnabled()</td>
</tr>
<tr>
<td></td>
<td>print(String)</td>
</tr>
<tr>
<td></td>
<td>println(String)</td>
</tr>
<tr>
<td></td>
<td>setOutputStream(OutputStream)</td>
</tr>
<tr>
<td>TraceManagerFactory</td>
<td>getModules()</td>
</tr>
<tr>
<td></td>
<td>registerModule(String)</td>
</tr>
<tr>
<td></td>
<td>registerModule(TraceModule)</td>
</tr>
<tr>
<td></td>
<td>registerModule(TraceModule, OutputStream)</td>
</tr>
</tbody>
</table>
## Cisco Trace Logging Interfaces

### Table A-17  Cisco Trace Logging Interfaces

<table>
<thead>
<tr>
<th>Cisco Trace Logging Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConditionalTrace</td>
<td>disable()</td>
</tr>
<tr>
<td></td>
<td>enable()</td>
</tr>
<tr>
<td>Trace</td>
<td>append(Object)</td>
</tr>
<tr>
<td></td>
<td>append(String)</td>
</tr>
<tr>
<td></td>
<td>getName()</td>
</tr>
<tr>
<td></td>
<td>isEnabled()</td>
</tr>
<tr>
<td></td>
<td>print(Object)</td>
</tr>
<tr>
<td></td>
<td>print(String)</td>
</tr>
<tr>
<td></td>
<td>print(String, Object)</td>
</tr>
<tr>
<td></td>
<td>print(String, String)</td>
</tr>
<tr>
<td></td>
<td>println(Object)</td>
</tr>
<tr>
<td></td>
<td>println(String)</td>
</tr>
<tr>
<td></td>
<td>println(String, Object)</td>
</tr>
<tr>
<td></td>
<td>println(String, String)</td>
</tr>
<tr>
<td></td>
<td>setDefaultMnemonic(String)</td>
</tr>
</tbody>
</table>
### Table A-17  Cisco Trace Logging Interfaces (continued)

<table>
<thead>
<tr>
<th>Cisco Trace Logging Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceManager</td>
<td>disableAll()</td>
</tr>
<tr>
<td></td>
<td>disableTimeStamp()</td>
</tr>
<tr>
<td></td>
<td>enableAll()</td>
</tr>
<tr>
<td></td>
<td>enableTimeStamp()</td>
</tr>
<tr>
<td></td>
<td>getConditionalTrace(String)</td>
</tr>
<tr>
<td></td>
<td>getConditionalTrace(String, String)</td>
</tr>
<tr>
<td></td>
<td>getName()</td>
</tr>
<tr>
<td></td>
<td>getOutputStream()</td>
</tr>
<tr>
<td></td>
<td>getSubFacilities()</td>
</tr>
<tr>
<td></td>
<td>getTraces()</td>
</tr>
<tr>
<td></td>
<td>getTraceWriter()</td>
</tr>
<tr>
<td></td>
<td>getUnconditionalTrace(String)</td>
</tr>
<tr>
<td></td>
<td>getUnconditionalTrace(String, String)</td>
</tr>
<tr>
<td></td>
<td>removeTrace(String)</td>
</tr>
<tr>
<td></td>
<td>removeTrace(Trace)</td>
</tr>
<tr>
<td></td>
<td>setOutputStream(OutputStream)</td>
</tr>
<tr>
<td></td>
<td>setSubFacilities()</td>
</tr>
<tr>
<td></td>
<td>setTraceWriter()</td>
</tr>
<tr>
<td>TRACETYPE</td>
<td>getTraceManager()</td>
</tr>
<tr>
<td></td>
<td>getTraceModuleName()</td>
</tr>
</tbody>
</table>
### Table A-17  Cisco Trace Logging Interfaces (continued)

<table>
<thead>
<tr>
<th>Cisco Trace Logging Interfaces</th>
<th>Method Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceWriter</td>
<td>close()</td>
</tr>
<tr>
<td></td>
<td>flush()</td>
</tr>
<tr>
<td></td>
<td>getEnabled()</td>
</tr>
<tr>
<td></td>
<td>print(String)</td>
</tr>
<tr>
<td></td>
<td>println(String)</td>
</tr>
</tbody>
</table>

| UnconditionalTrace            |                      |
# CTI Error Codes

## Table B-1  CTI Error Codes

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFWDALL_ALREADY_OFF</td>
<td>Attempt to turn off CFWALL while it is already off</td>
</tr>
<tr>
<td>CFWDALL_ALREADY_SET</td>
<td>Attempt to set CFWALL while it is already set</td>
</tr>
<tr>
<td>CFWDALL_DESTN_INVALID</td>
<td>Attempt to CFWALL to an invalid destination</td>
</tr>
<tr>
<td>COMMAND_NOT_IMPLEMENTED_ON_DEVICE</td>
<td>Internal call processing error: device does not support the command</td>
</tr>
<tr>
<td>CONFERENCE_ALREADY_PRESENT</td>
<td>Attempt to conference a party that is already in conference</td>
</tr>
<tr>
<td>CONFERENCE_FAILED</td>
<td>Conference complete was not successful</td>
</tr>
<tr>
<td>CONFERENCE_FULL</td>
<td>All conference bridges are busy</td>
</tr>
<tr>
<td>CONFERENCE_INACTIVE</td>
<td>Attempt to complete conference while consult conference is not active</td>
</tr>
<tr>
<td>CONFERENCE_INVALID_PARTICIPANT</td>
<td>Trying to conference to self or an invalid participant</td>
</tr>
<tr>
<td>Error Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CTIERR_ASSOCIATED_LINE_NOT_OPEN</td>
<td>Command issued on a line that must be open</td>
</tr>
<tr>
<td>CTIERR_CALL_AREady_EXISTS</td>
<td>Another call already exists on the line</td>
</tr>
<tr>
<td>CTIERR_CALLHANDLE_NOTINCOMINGCALL</td>
<td>Attempt to answer a call that either does not exist or is not in the correct state</td>
</tr>
<tr>
<td>CTIERR_CALLHANDLE_UNKNOWN_TO_LINECONTROL</td>
<td>Attempt to redirect call that was unknown to line control</td>
</tr>
<tr>
<td>CTIERR_CANNOT_OPEN_DEVICE</td>
<td>Device open failed because the associated device is shutting down (unregistering)</td>
</tr>
<tr>
<td>CTIERR_CANNOT_TERMINATE_MEDIA_ON_PHONE</td>
<td>Media cannot be terminated by an application when the device has a physical phone (the phone always terminates the media)</td>
</tr>
<tr>
<td>CTIERR_CLUSTER_LINK_FAILURE</td>
<td>Link failed to one of the call managers in the cluster (network error)</td>
</tr>
<tr>
<td>CTIERR_COMMAND_NOT_IMPLEMENTED_ON_DEVICE</td>
<td>Device does not support the command</td>
</tr>
<tr>
<td>CTIERR_DB_ERROR</td>
<td>Device query contained an illegal device type</td>
</tr>
<tr>
<td>CTIERR_DB_ILLEGAL_DEVICE_TYPE</td>
<td>No longer used</td>
</tr>
<tr>
<td>CTIERR_DB_NO_MORE_DEVICES</td>
<td>No longer used</td>
</tr>
<tr>
<td>CTIERR_DEVICE_NOT_OPEN</td>
<td>Attempt to open either a line on a device not open, or a device that must already be registered (i.e., an IP phone device)</td>
</tr>
<tr>
<td>CTIERR_DIRECTORY_LOGIN_FAILED</td>
<td>Login to the directory server failed when opening the provider</td>
</tr>
</tbody>
</table>
### Table B-1  CTI Error Codes (continued)

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTIERR_HOLDFAILED</td>
<td>Hold was rejected by line control or call control</td>
</tr>
<tr>
<td>CTIERR_ILLEGAL_CALLINGPARTY</td>
<td>Attempt to originate call using a calling party that is not on the device</td>
</tr>
<tr>
<td>CTIERR_ILLEGAL_CALLSTATE</td>
<td>Line is not in a legal state to invoke the command</td>
</tr>
<tr>
<td>CTIERR_ILLEGAL_HANDLE</td>
<td>Handle is unknown to the system</td>
</tr>
<tr>
<td>CTIERR_ILLEGAL_MESSAGE_FORMAT</td>
<td>Device registration failed due to incorrect media capability</td>
</tr>
<tr>
<td>CTIERR_INTEGRITY_DTMFDIGITS</td>
<td>Play DTMF request failed because it is an invalid DTMF digit</td>
</tr>
<tr>
<td>CTIERR_LINECONTROL_FAILURE</td>
<td>Line control refuses to let a new call be initiated because of its state (probably bug)</td>
</tr>
<tr>
<td>CTIERR_MEDIA_ALREADY_TERMINATED</td>
<td>Attempt to terminate on a media device that already has media terminated</td>
</tr>
<tr>
<td>CTIERR_NOT_INITIALIZED</td>
<td>Attempt to open a provider before CTI initialization completes</td>
</tr>
<tr>
<td>CTIERR_OPERATION_FAILED_QUIETCLEAR</td>
<td>Feature unavailable for this call due to temporary failure</td>
</tr>
<tr>
<td>CTIERR_PROVIDER_ALREADY_OPEN</td>
<td>Attempt to reopen a provider</td>
</tr>
<tr>
<td>CTIERR_PROVIDER_NOT_OPEN</td>
<td>Attempt to issue a CTI command before the provider was open</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_CALL_TABLE_FULL</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_DESTINATION_BUSY</td>
<td>Redirect destination is busy</td>
</tr>
<tr>
<td>Error Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_DESTINATION_OUT_OF_ORDER</td>
<td>Redirect destination is out of order</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_DIGIT_ANALYSIS_TIMEOUT</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_DOES_NOT_EXIST</td>
<td>Attempt to redirect a call that does not exist or is not longer active</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_INCOMPATIBLE_STATE</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_MEDIA_CONNECTION_FAILED</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_NORMALCLEARING</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_ORIGINATOR_ABANDONED</td>
<td>Far end hung up on the call being redirected</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_PARTY_TABLE_FULL</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_PENDING_REDIRECT_TRANSACTION</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_PROTOCOL_ERROR</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_UNKNOWN_DESTINATION</td>
<td>Attempt to redirect to an unknown destination</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_UNKNOWN_ERROR</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_UNKNOWN_PARTY</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALL_UNRECOGNIZED_MANAGER</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_REDIRECT_CALLINFO_ERR</td>
<td>Internal error returned from call control</td>
</tr>
</tbody>
</table>
### CTI Error Codes (continued)

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTIERR_REDIRECT_ERR</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>CTIERR_RETRIEVEFAILED</td>
<td>Retrieve was rejected by line control or call control</td>
</tr>
<tr>
<td>CTIERR_SSAPI_NOT_REGISTERED</td>
<td>Redirect command was issued when the internal supporting interface was not initialized; either CTI has not yet finished its initialization of and internal error occurred.</td>
</tr>
<tr>
<td>CTIERR_TIMEOUT</td>
<td>No longer used</td>
</tr>
<tr>
<td>CTIERR_TRANSFERFAILED</td>
<td>Transfer failed (probable cause is one of the call legs was hung up or disconnected from the far end)</td>
</tr>
<tr>
<td>CTIERR_TRANSFERFAILED_CALLCONTROL_TIMEOUT</td>
<td>Expected response from call control not received during a transfer</td>
</tr>
<tr>
<td>CTIERR_TRANSFERFAILED_DESTINATION_BUSY</td>
<td>Attempt to transfer to a busy destination</td>
</tr>
<tr>
<td>CTIERR_TRANSFERFAILED_DESTINATION_UNALLOCATED</td>
<td>Attempt to transfer to a directory number that is not registered</td>
</tr>
<tr>
<td>CTIERR_TRANSFERFAILED_TRANSFER_ALREADY_OUTSTANDING</td>
<td>Existing transfer is still in progress</td>
</tr>
<tr>
<td>CTIERR_UNDEFINED_LINE</td>
<td>Line was specified that was not found on the device</td>
</tr>
<tr>
<td>CTIERR_UNKNOWN_GLOBAL_CALL_HANDLE</td>
<td>No longer used</td>
</tr>
<tr>
<td>CTIERR_UNRECOGNIZABLE_PDU</td>
<td>QBE protocol error (bug)</td>
</tr>
<tr>
<td>DARES_INVALID_REQ_TYPE</td>
<td>Internal call processing error: DaRes invalid request type</td>
</tr>
<tr>
<td>DATA_SIZE_LIMIT_EXCEEDED</td>
<td>XML data object size is bigger than allowed</td>
</tr>
</tbody>
</table>
### Table B-1  CTI Error Codes (continued)

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVICE_OUT_OF_SERVICE</td>
<td>Device is out of service</td>
</tr>
<tr>
<td>DIGIT_GENERATION_ALREADY_IN_PROGRESS</td>
<td>Digit generation is already in progress</td>
</tr>
<tr>
<td>DIGIT_GENERATION_CALLSTATE_CHANGED</td>
<td>Call state invalid to continue</td>
</tr>
<tr>
<td>DIGIT_GENERATION_WRONG_CALL_HANDLE</td>
<td>Call handle is invalid and call may be already gone</td>
</tr>
<tr>
<td>DIGIT_GENERATION_WRONG_CALL_STATE</td>
<td>Call state is not valid to generate digits</td>
</tr>
<tr>
<td>DIRECTORY_TEMPORARY_UNAVAILABLE</td>
<td>Directory is temporarily unavailable</td>
</tr>
<tr>
<td>INVALID_LINE_HANDLE</td>
<td>Attempt to perform a line operation on an invalid line handle</td>
</tr>
<tr>
<td>LINE_INFO_DOES_NOT_EXIST</td>
<td>Line information does not exist in database DbDNResponse</td>
</tr>
<tr>
<td>LINE_NOT_PRIMARY</td>
<td>Internal error returned from call control</td>
</tr>
<tr>
<td>MAX_NUMBER_OF_CTI_CONNECTIONS_REACHED</td>
<td>The maximum number of CTI connections reached; may need to reconfigure the maximum number of CTI connections or logout of some applications to be able to log in</td>
</tr>
<tr>
<td>MSGWAITING_DESTN_INVALID</td>
<td>Attempt to set message waiting lamp for an invalid DN; Message Waiting Destination not found</td>
</tr>
<tr>
<td>OPERATION_NOTA_AVAILABLE_IN_CURRENT_STATE</td>
<td>Feature operation is not available in the current state</td>
</tr>
<tr>
<td>PROVIDER_CLOSED</td>
<td>Attempt to close provider while it is already closed</td>
</tr>
<tr>
<td>PROTOCOL_TIMEOUT</td>
<td>Internal error returned from call control</td>
</tr>
</tbody>
</table>
Table B-1  CTI Error Codes (continued)

<table>
<thead>
<tr>
<th>Error Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRIEVEFAILED_ACTIVE_CALL_ON_LINE</td>
<td>Error in retrieving held call; call may be already dropped</td>
</tr>
<tr>
<td>TRANSFER_INACTIVE</td>
<td>Attempt to complete transfer while consult transfer is not there</td>
</tr>
</tbody>
</table>
INDEX

A
Actor.java
Cisco JTAPI examples 3-5
addresses and terminals
relationship 1-6
unobserved 1-7
alarm services 1-10

C
call forward setting 1-13
call park 1-13
reminder 1-14
retrieval 1-14
CiscoCallID 1-8
CiscoConnectionID 1-8
Cisco JTAPI 1-2
  install internationalization 1-15
Cisco JTAPI Examples
  MakeCall.java 3-2
Cisco JTAPI examples
  Actor.java 3-5
makecall.java 3-2
Originator.java 3-10
Receiver.java 3-14
running makecall 3-21
StopSignal.java 3-16
Trace.java 3-17
TraceWindow.java 3-18
CiscoJtapiExceptions 1-15
clear calls interface 1-15
conference
Cisco extensions 1-23
events 1-25
scenarios 1-24
transfer and conference enhancement 1-26
connection 1-7

D
device recovery 1-15
CTI ports 1-16
CTIRoutePoint 1-16
phones 1-16
directory change notification 1-17

I
invoking makecall 3-21
Index

J

JTAPI parameters
application control 1-10
Jtapi.ini 1-11

M

MakeCall.java
Cisco JTAPI Examples 3-2
makecall.java 3-2
media
CiscoMediaTerminal 1-31
endpoint model 1-28
initialization 1-29
media termination extensions 1-27
payload and parameter negotiation 1-28
payload selection 1-29
receive channel allocation 1-30
receiving and responding to media flow events 1-35
starting transmission and reception 1-30
stopping transmission and reception 1-31

N

new and changed information xxvii

O

Originator.java
Cisco JTAPI examples 3-10
overview 1-1

R

Receiver.java
Cisco JTAPI examples 3-14
redirect 1-37
redundancy 1-42
CTManager failure 1-48
CTManagers 1-46
heartbeats 1-49
invoking CTManager 1-47
server failure 1-44
related documentation xxx
required software xxx
ringer
enable or disable 1-17
routing 1-38
running makecall.java 3-21

S

StopSignal.java
Cisco JTAPI examples 3-16
Index

T

TerminalConnection 1-8
threaded callbacks 1-8
Trace.java
  Cisco JTAPI examples 3-17
TraceWindow.java
  Cisco JTAPI examples 3-18
transfer and conference
  conference 1-22
transfer and conference
  consult without media 1-26

x

xsi object pass through 1-53