Agent Interaction SDK Java Developer Guide

Switch Facilities

12/16/2025

Contents

Switch Facilities

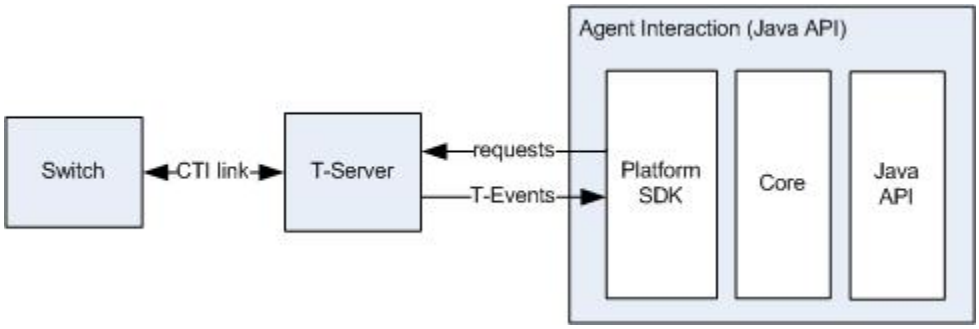
This chapter describes the switch facilities provided by the Agent Interaction (Java API).

Switch Design

The Agent Interaction (Java API) provides easy access to available switch features and data, and facilitates development when different switches are involved.
To ensure both data persistence and consistency, the core library manages connections with Genesys components and deals with incoming information by maintaining corresponding objects.
So forth, the Agent Interaction (Java API) uses a voice state model to manage components related to voice features (such as outbound, callback, and so on), and CTI objects corresponding to the use of particular switches across their T-Servers.
As far as they are able, state machines guarantee that the voice state model is coherent with other Genesys components (for example, multimedia solutions) across supported switches. This facilitates the integration of these services and applications. In contrast, developers using the Platform SDK must build their own state machines, which increases the integration complexity.

T-Server Connections

The AIL library core does not directly connect to any switches, but rather to the T-Servers that manage the switches. The AIL core library integrates with the Platform SDK to communicate with the T-Server that drives the switch. Therefore, the AIL's CTI features are limited to T-Server CTI features: The AIL library provides you only with what the T-Servers can perform.



AIL Driving the T-Server via the Platform SDK

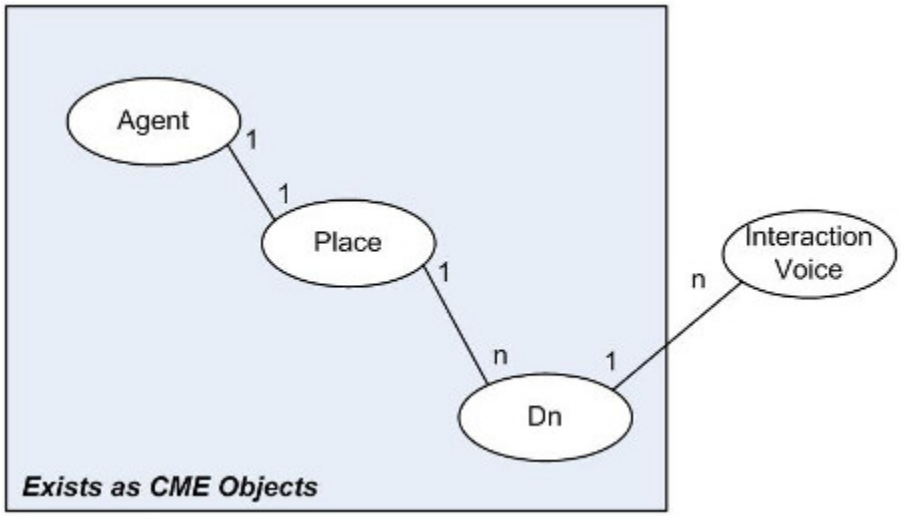
Through the Platform SDK, the AIL core library sends requests to the T-Server in order to perform agent actions (See Calls Mapping.) The T-Server manages its connection with the switch and generates requests to drive the switch
Then, the T-Server notifies the AIL core library with TEvents. The AIL core library updates its model, then notifies listeners with an event built from information contained in the TEvents.

> **Warning**
> The AIL library makes no assumption about the success or failure of any request processed through the API to the T-Server and only updates with TEvents.

Voice Model

The voice model is consistent with the models defined for other media. Prior to any voice manipulation, your application must log in an agent on a voice media (a DN) which is available in a `Place`. Through this logged DN, you get interactions that enable voice actions. Managing the interaction on the switch involves manipulating the `Dn` and `InteractionVoice` interfaces.

The voice model that relies on the main AIL classes—`Agent`, `Place`, and `Dn`—is presented in The Voice Model.



The Voice Model

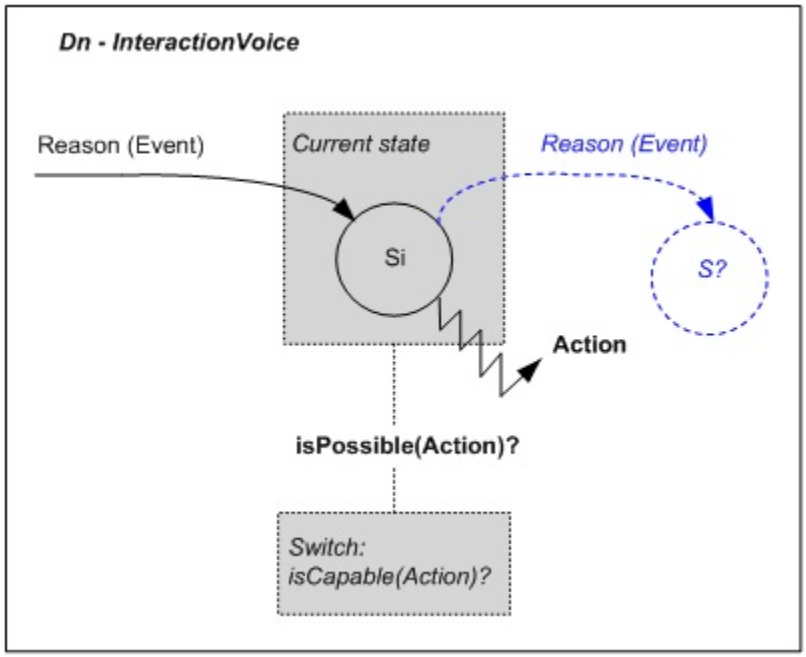
Voice State Model

The `Switch` interface accesses general information, that do not change at runtime, in contrast to the `Dn` and `InteractionVoice` instances that agent actions affect. For these two last classes, the AIL library `provides you with a voice state model, defined as follo` ws:

- `States`: These are the results of actions. Your application is notified of these actions as the switch performs them. The current state affects the range of actions that your application can perform.
- Transitions: These depend on the success of the last action. They are performed according to the last notified TEvent and are labeled according to the TEvent name.

States

Internally, the AIL core library manages both `Dn` and `InteractionVoice` objects through their own state machines. These and other objects inherit from the `Possible` interface, which offers the `isPossible()` method to test whether an action can be performed from the current state.

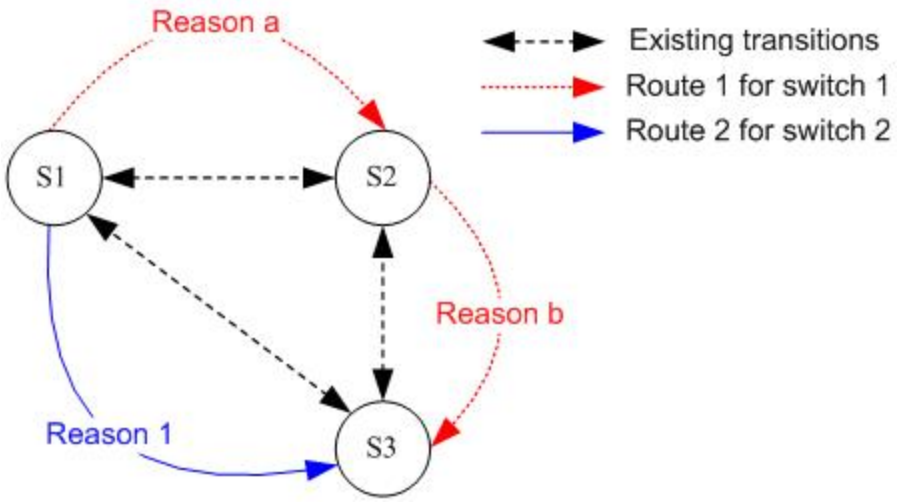


Possibility of Performing an Action on the Switch

This figure shows the `isPossible()` method dependencies. This method takes into account the current state of the object, and whether the switch is capable of a particular action, to determine the action's possibility with respect to current state. For details, see Determine Availability of CTI Features.

Transitions

Because the TEvent is an *acknowledge* action coming from the T-Server, the AIL core library first updates its model, then it builds the corresponding `DnEvent` or `InteractionEvent` events that notify your client application. These events' reason correspond to TEvent names.
The T-Server call flow differs from one switch to another: it is affected by switches, switch settings, and T-Server settings. Therefore, the library cannot impose uniformity on the transition sequences. The routes in state models are switch-dependent, as illustrated in Reasons and Routes.



Reasons and Routes

The figure shows that from one switch to another, the TEvent sequence is different and is associated with different reasons and routes in the state models. To cite a concrete example, an application monitoring a `Routing Point` has a state sequence that is different on Avaya G3 versus Nortel Meridian switches.

> ### Important
> You should design your applications with respect to object status rather than event reasons. The state model ensures coherence and reliability, whereas event reasons are strongly switch-specific.

> ### Warning
> Do not make any assumptions about incoming events. Refer to the T-Server Deployment Guide for your environment, to the Genesys 7 Events and Models Reference Manual for model details, and to the Agent Interaction SDK 7.6 Java API Reference for the full lists of reference material relating to the Agent Interaction (Java API).

Switch and DN Management

Now that you have been introduced to the switch implementation in Agent Interaction (Java API), it is time to outline the consideration you will need to work with switches.
First, in order to log in properly on DNs, you must learn about the DN Consolidation, that explains the

DN consolidation through the `Dn` interface. It presents how you identify and access consolidated DNs defined for a given switch.
Then, before you make calls to voice features, you check whether these calls are available, as explained in Determine Availability of CTI Features.
After you checked the feature is available, you can perform a method call. This call is mapped with the Platform SDK, as detailed in Calls Mapping.
As the Agent Interaction Java API makes no assumption about action result, to get acknowledge of successful actions, you listen to events, as explained in Event Flow.

DN Consolidation

Regardless of the DN types that the switch handles, the DN consolidation model provides a single `Dn` interface with a unique DN ID, that a voice interaction reaches through its callable number, as detailed in the following subsections.

DN Types

Switch DNs' types—for example, ACD position or extension—are mostly switch-specific. This affects the use of such features as login, logout, ready actions, and so on.

The AIL library provides consolidation of the `Dn` object and its model so that you do not have to write the code to manage this in your application. In all cases, the library exposes a single `Dn`, even if the configuration of a place requires more DNs in the Configuration Layer according to the underlying switch.

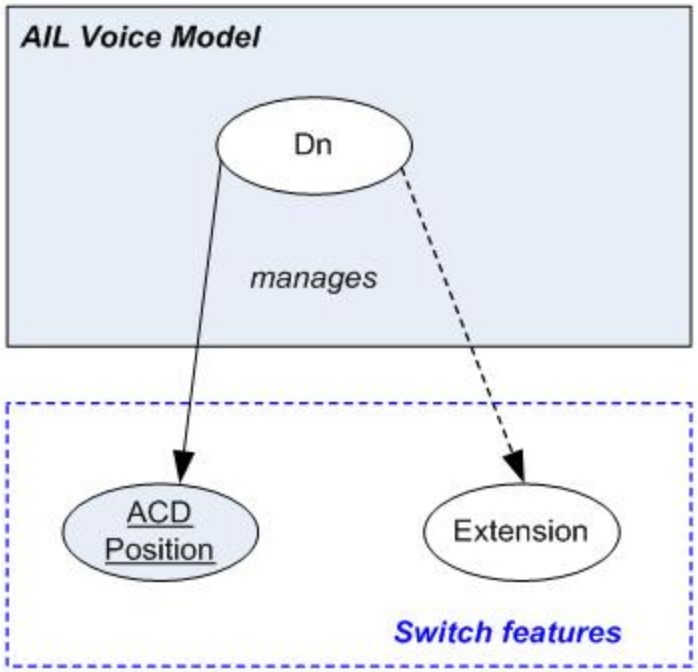
To find the required configuration for each switch, and thus find which DN type is visible, refer to the *Interaction SDK 7.6 Java Deployment Guide.*

> ### Warning
> You must respect this DN consolidation model if your applications are to handle DNs properly.

For example, to work in regular mode with an A4400 switch, the AIL:

- Registers the `ACDPositio` n and `Extension.`
- Exposes a single `Dn` object.

The Agent Interaction (Java API) presents a single DN to transparently manages the requests to hidden ACDPositions and Extensions. For A4400, the DN ID exposed in the API is the ACDPosition, as shown below.



A4400 - DN Consolidation

As another example, to work in substitute mode on an A4400 switch, your application must manage activities of an `extension` on a `Place` . When no agent is logged in, the extension is visible and the ACD position DN is not visible. When an agent is logged in, the ACD position DN is visible in the place and the extension is not visible in the place. For details about the corresponding events, refer to DN Events.

DN ID

The DN ID is used to retrieve the corresponding `Dn` instance through the `AilFactory.getDn()` method.
In this product's 6.5 releases, the DN ID used to be the Configuration Layer's DN callable number. This restriction did not allow two distinct switches to declare two identical DN numbers. To avoid any ambiguity and to ensure unique DN IDs, a DN ID is now defined in accordance with the following rule:`<DN_CME_Name>@<switch_name>`

> ### Important
> Set the `enabled` option to `false` in the `dn-at-switch` section to disable this rule and retrieve DN IDs as defined in 6.5 releases.

Callable Number

The callable number is the number that your application must use to reach a DN when, for instance,

making a call. Depending on the switch, the DN number declared in the Configuration Layer might not be the number to dial for call actions, such as make call, transfer, or conference. For example, you must remove some leading numbers to reach a Nortel DMS 100's DN.
The number to dial can also depend on the DN status. For example, if an agent is logged into the DN, you have to dial the agent ID.
Finally, because of the consolidation model, some DNs can be hidden but still be the real ones to dial instead of their visible counterpart. Once again, the callable number of a visible DN properly returns the hidden DN's number.
To determine the callable number of a `Dn` instance at runtime, call the `Dn.getCallableNumber()` method.

> **Important**
> Genesys recommends using callable numbers to access CTI features.

DN Activation

To perform telephony actions and receive telephony events for a DN on a T-Server, the library uses an internal `Dn` object, represented in the Agent Interaction (Java API) by the `Dn` interface.

Typically, when an agent uses your application to log in, the library creates a new `Dn` object for the agent's DN, but also creates a new monitoring session for the new `Dn` object. The library uses the monitoring session to register the `Dn` on its T-Server and then to monitor activity through the life of the `Dn` object. Usually, the DN's monitoring session is released at garbage collection time.

Typically, when developing a server application, it is possible that after one agent logs out, another agent may log in to work with the same DN. Your application can keep a `Dn` object alive in the time period between one agent's logout and another agent's login, but in that case, monitoring could cause unwanted interactions.

To kill a monitoring session for a DN, use the `Dn.unactivate()` method, which unregisters the DN on the T-Server and kills the monitoring session.

Later, at the time another agent is about to log in to start work with the same DN, the internal `Dn` object is still alive but inactive because it is not registered to its T-Server and it has no monitoring session. Before the agent can log in, your application must create a new session that re-registers the DN on the T-Server and lets the library perform actions and receive events. To create a new monitoring session, call the `Dn.activate()` method. This registers the DN on the T-Server, allowing the new login, and instantiates a new monitoring session for that `Dn` object.

> **Important**
> By default, registering and unregistering actions are handled by the library. If your client application calls the `activate()` or `unactivate()` method, your application is responsible for managing the monitoring session of the relevant DN.

The `activate()` and `unactivate()` methods are documented in the *Agent Interaction 7.6 Java API Reference* as members of the `AbstractDn` interface, which is a Superinterface for the `Dn` interface.

Determine Availability of CTI Features

After your agent has successfully logged in on a DN, your application will access CTI features only if these features are available on the DN's switch. Examples of such possibly unavailable features on a switch are `Transfer`, `Conference`, `Callback`, and `Holding`. Refer to your T-Server documentation to see which features are available on a given switch.
The Agent Interaction (Java API) is agent-oriented. This means that for a logged in agent at runtime, you are provided with all possible CTI features, and even methods to determine which features are available. These methods indicate switch *capabilities* and *possibilities,* as detailed in the following subsections.

Capability

You access capability through the `Switch interface`, which has the following characteristics:

- An instance of the `Switch` class is a static object.
- Access to a `Switch` object is given by the `Dn.getSwitch()` method, or `AilFactory getSwitch(java.lang.String name).`

The `Switch.isCapable()` method takes into account the switch's features and returns the switch's capability to perform the `InteractionVoice.Action` during the entire runtime session. The result is independent from the voice DN's current state.
Capability testing enables your application to determine what features are available on the switch and behave appropriately—for example, to show or not show buttons, menus, and so on.
For example, the `MultipartyVoiceInteraction` example tests transfer capabilities to determine whether it should display or not buttons for the transfer and conference features, as show in the following code snippet:

```
if(sampleDn instanceof Dn )
{
        Switch theSwitch = sampleDn.getSwitch();
        if (theSwitch != null) {
                switchCanDoSingleStep =
theSwitch.isCapable(InteractionVoice.Action.SINGLE_STEP_TRANSFER)
                || theSwitch.isCapable(InteractionVoice.Action.SINGLE_STEP_CONFERENCE);
                switchCanDoMuteTransfer =
theSwitch.isCapable(InteractionVoice.Action.MUTE_TRANSFER);
                switchCanDoDualStep =
theSwitch.isCapable(InteractionVoice.Action.INIT_TRANSFER)
                || theSwitch.isCapable(InteractionVoice.Action.CONFERENCE);
        }
}
singleStep.setVisible(switchCanDoSingleStep);
muteTransfer.setVisible(switchCanDoMuteTransfer);
dualStep.setVisible(switchCanDoDualStep);
```

Possibility

According to the current state of the objects (`Dn or Interaction`), a feature may be temporarily unavailable, although the switch is capable of performing the corresponding action. For instance, in most cases, if your agent is not logged in on a DN of the switch, your agent will not be able to create a new voice interaction with the corresponding `Dn` interface.
After the agent will be logged in, the Agent Interaction Java API will get an event that will change its

DN status, so that creating a voice interaction will become possible.
Possibility, managed through the `Possible` superinterface with the `isPossible()` method, is applied to actions that you wish your application to perform. To determine an action possibility, the `isPossible()` method concatenates:

- The object's current state, as calculated by the AIL.
- The switch capability to perform the action.

This is illustrated in Possibility of Performing an Action on the Switch.
Genesys recommends using this `isPossible()` method to determine feature accessibility each time an event occurs.

> **Warning**
>
> The `isPossible()` method does not reflect the availability of features that depend on DN type, T-Server options, or the switch environment (ACD, Genesys Routing, proprietary call distribution, and so on.)

A GUI application should use the `isPossible()` method to enable or disable buttons according to events. The following code snippet is extracted from this method and manages three radio buttons, enabling the agent to select the transfer to perform:

```
//The complete button should be enabled if a transfer or a conference
//can be completed
boolean complete = sampleInteraction.isPossible(InteractionVoice.Action.COMPLETE_TRANSFER)
   || sampleInteraction.isPossible(InteractionVoice.Action.COMPLETE_CONFERENCE);
completeButton.setEnabled(complete);
```

> **Important**
>
> Remember that the current state changes with events. Your event handlers should continually validate the availability of features.

Calls Mapping

The AIL core library handles switch specifics internally to allow easy, consistent management of voice features across all switches that AIL supports. Refer to the *Interaction SDK 7.6 Java Deployment Guide* for the list of supported switches.
When you make calls to voice methods, the AIL core library transparently takes over the required calls to Platform SDK functions, regardless of the switch type.

For Platform SDK users, the mapping of the Platform SDK features is straightforward because the naming convention is similar to that for Platform SDK functions. By contrast, building an application using the Platform SDK would require that you write code to deal with switch specifics in your function calls.

One example is the DMS100 switch and its specific implementation of a `makeCall()` method. Using

the Platform SDK, your application must first create a RequestMakeCall object and then the RequestAnswerCall object, as illustrated in the following code snippet. If it does not do both, the agent must manually go off hook to begin the call.

```
//requesting a MakeCall to the TServer using the Platform SDK
server.send( RequestMakeCall.create(
                _dn,
                _otherdn,
                _make_call_type,
                _location,
                _userdata,
                _reasons,
                _textensions);
//....
//This is a DMS100 switch: requesting an AnswerCall
server.send( RequestAnswerCall.create(
                _dn,
                NULL_CONNECTION_ID,
                _reasons,
                _textensions);
//...
```

The benefit of hiding these switch-specific interventions is that your code is less complex and your feature management is easier, available, and works for all switches supported by AIL. The AIL code corresponding to the previous Platform SDK code snippet is:

```
// Dial the call
voice.makeCall( myDn.getCallableNumber(), //callable number to dial
  _location, //location
  InteractionVoice.MakeCallType.REGULAR, //Type
  _userdata, //User data
  _reasons, //Reasons
  _textensions);//Textensions
//...
```

Although the Agent Interaction (Java API) simplifies switch management, you can still tune method calls with switch-specific parameters. Refer to Switch Tuning for further details.

Event Flow

The AIL core library makes no assumption of the method call's result. As explained in Voice State Model, after the request succeeds, the AIL core library will get a TEvent that will generate a DnEvent or InteractionEvent event (according to the called method.)
In case the method call fails, you should get an exception. Refer to the *Agent Interaction 7.6 API Reference* for further details about method exceptions.

> ### Important
>
> Because a Place reflects **all** Dn and Interaction events, Genesys recommends that you use a PlaceListener rather than using separate listeners for each Dn orInteraction.

DN Events

Through the `Place.handleDnEvent()` method, your application gets `DnEvent` events, that is information about successful login, logout, ready, not ready actions performed on the consolidated DNs of this place.
Through these events, you will get accurate information about DN status, in particular for some DNs that become invisible due to the consolidation (as explained in DN Consolidation).
As an example, to work in substitute mode on an A4400 switch, your application must manage activities of an `extension` on a `Place`. When an agent logs in successfully, the `Place.handleDnevent()` method gets two events:

- `dnRemoved` provides notification that the `extension` is no longer visible.
- `dnAdded` provides notification that a `Dn` of type `ACD position` is now visible.

Successful agent login generates a login event; the login event and all subsequent event and request activities occur with respect to the ACD position DN, not the extension.
When the agent logs out, logout is performed through the ACD position DN, and upon successful logout, the `Place.handleDnevent()` method gets four events:

- An event carrying notification of successful logout.
- A `dnRemoved` event carrying notification that the ACD position DN is no longer visible.
- A `dnAdded` event with notification that the extension is now visible.
- An event carrying notification of the extension status.

When no agent is logged in, the extension is visible and the ACD position DN is not visible. When an agent is logged in, the ACD position DN is visible in the place, and the extension is not visible in the place.

Single-Step Rollover to Mute Transfer

The Agent Interaction (Java API) provides you with a hidden mute transfer in the `singleStepTransfer()` method:

- If the switch is not capable of performing the single-step transfer, AIL transparently performs a mute transfer instead.
- The `isCapable(InteractionVoice.Action.SINGLE_STEP_TRANSFER)` call takes into account the hidden mute capability.
- The `isPossible(InteractionVoice.Action.SINGLE_STEP_TRANSFER)` call takes into account the hidden mute possibility.

Even `if true single-step transfer` is not available, the capability and possibility results indicate whether your application can perform the mute transfer instead. So, you can rely on this result to perform your transfer with the `singleStepTransfer()` method.

Switch Tuning

To fine-tune your application according to your switch, the Agent Interaction (Java API) provides you with TExtensions and workmodes.

> **Warning**
> Managing some switch-specific data implies a dependency on the corresponding particular switch.

TExtensions

`TExtensions` are `Map` data structures that take into account switch-specific features and information that cannot be described in a request parameter. The library transmits them as parameters in Platform SDK calls used to tune T-Server operations.
To get details about the list of extensions associated with a switch, refer to the corresponding T-Server documentation.
`TExtensions` exist for both `Dn` and `InteractionVoice` interfaces. You get them by calling the `getTExtensions()` methods. Refer to the *Genesys Platform SDK 7.6 Developer's Guide* for further information.

In Method Calls

To add TExtensions in a method call, you just need to create a `Map` of key-value pairs and pass it as parameter in your voice method call.
For example, if you are working on an application dedicated to the G3 switch, you may want to specify the `Trunk` TExtension for the `makeCall()` method, as shown in the following code snippet.

```
String myTrunk = "...";
HashMap myTExtension = new HashMap();
myTExtension.put("Trunk", myTrunk);
voice.makeCall( DN,
 null,
 InteractionVoice.MakeCallType.REGULAR,
 null,
 null,
 myTExtension);
```

> **Important**
> TExtensions that apply only to a given switch are described in the corresponding individual T-Server manual.

In Events

Both DnEvent and InteractionEvent events propagate TExtensions, that are switch-specific TEvent Extensions. These TExtensions are copied from the TEvent and are additional data provided for switch-specific interventions.
You can retrieve these TExtensions in a Map returned by calling InteractionEvent.getTEventExtensions() or DnEvent.getTEventExtensions(). As TExtensions are switch-specific, refer to your T-Server documentation to learn more about its TExtensions.
The following code snippet shows how to deal with TExtension Maps.

```
// Implementation of the Agent.HandleInteractionEvent() method
public void handleInteractionEvent(InteractionEvent _ie) {
        //...
        //Retrieval of the map containing the TExtensions

        Map myTExtensions = ie.getTEventExtensions();
        //Testing if there is TExtension attached in the event
        if (mTExtensions != null) {
                // Retrieving an iterator for the key set
                Iterator it = mTExtensions.keySet().iterator();
                while (it.hasNext() ) {
                        String key = (String) it.next();
                        System.out.println("Key: "+key+" Value: "+mTEventExtensions.get(key));
                }
        }
        //...
}
```

> ### Warning
> TExtensions can be optional in a TEvent, so the library propagates them only if they exist. Refer to your T-Server documentation for more information.

Workmodes

Like the Platform SDK, the Agent Interaction (Java API) provides you with dedicated methods to handle workmodes. Workmodes are used in agent events to provide more detailed information about an agent's actual state.
For example, the Manual in workmode for a log action indicates that the agent must validate the action manually on the phone. The After Call Work workmode indicates the agent is still working on the last call.
Workmodes are identified by the Dn.Workmode class. Refer to the Javadoc API Reference to get the list of workmodes taken into account by the API.
Workmodes can be specified in the parameters of dedicated methods related to Dn.Action. You can use them in the Agent, Place, and Dn interfaces for calls to their respective login(), logout(), ready(), and notready() methods.

> **Important**
>
> If you use a workmode in a method call performed from an `Agent` or `Place` object, all the associated voice DNs can be affected by the workmode that you specify.

Workmodes

To test available workmodes, the Agent Interaction (Java API) provides you with specific methods in the `Switch` and `Dn` interfaces:

- `Switch.isWorkmodeCapable(Dn.Workmode)` —`true` if the switch supports the DN workmode.
- `Dn.isWorkmodePossible(Dn.Workmode)` —`true` if the DN workmode is available for the next action on the `Dn` .
- `Dn.getPossiblebleWorkmodes()`: returns, in a table of `boolean` values, the availability of the different workmodes for the next action on the `Dn`.

You can use these possibility and capability tests in the same manner as the standard tests, which are described in Capability and Possibility.

> **Important**
>
> The provided workmode tests reflect only the T-Server's workmode availability. AIL does not take into account any action to perform.

The following code snippet shows how to use the `Dn.isWorkmodePossible()` method:

```
//...
if(mDn.isWorkmodePossible(Dn.Workmode.NO_CALL_DISCONNECT)==true) {
   mDn.ready(mQueue, Dn.Workmode.NO_CALL_DISCONNECT, null, null);
}
//...
```

> **Warning**
>
> This capability test is not processed by the T-Server and does not take into account any T-server or switch settings. Refer to your T-Server documentation to enable your workmodes.

After Call Work

The API provides additional `afterCallwork()` methods in the `Agent`, `Place`, and `Dn` classes specific to the After Call Work mode. You can call these methods without specifying any workmode, as shown in

the following code snippet:

```
myAgent.afterCallwork(null, null, null);
```

The `afterCallwork()` methods are equivalent to a `notReady()` call with the workmode `AFTER_CALL_WORK` passed as a parameter, as shown in this code snippet:

```
myAgent.notready(null, Dn.Workmode.AFTER_CALL_WORK,null,null);
```

The `afterCallwork()` methods enable you to put the concerned DNs into the associated status: `Dn.status.AFTERCALLWORK`. This status is an extension of the `NotReady` state: it means that the agent is not ready to receive another call, because he or she is working on another task.

> ### Warning
> If the `After Call Work` workmode is not supported by the underlying switch, `afterCallwork()` methods are unavailable.