

# **Oracle® Essbase Administration Services**

## **Developer's Guide**

RELEASE 11.1.2

Administration Services Developer's Guide, 11.1.2

Copyright © 2001, 2011, Oracle and/or its affiliates. All rights reserved.

Authors: EPM Information Development Team

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited. The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

**U.S. GOVERNMENT RIGHTS:**

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

# Contents

---

<b>Documentation Accessibility</b> .....	5
<b>Chapter 1. Introduction</b> .....	7
About Administration Services .....	7
About Java Plug-in Components .....	8
Requirements for Using Administration Services Java Plug-ins .....	9
Prerequisite Knowledge .....	9
Framework Concepts .....	10
Packaged APIs for Administration Services .....	10
Administration Services Java Packages .....	10
Example Classes .....	10
About the Sample Code in this Guide .....	10
<b>Chapter 2. Writing Client Plug-ins</b> .....	11
Access Point for Plug-ins .....	11
Class Packages .....	11
How the Client Locates Plug-ins .....	13
Creating the Miscellaneous Handler Class .....	13
Adding Functionality .....	14
Semantic Rules .....	14
Adding a Branch to the Enterprise Tree .....	14
Adding Children to Other Tree Nodes .....	16
Permitting Plug-ins To Add Children To Your Tree Nodes .....	17
Adding Context Menu Items To Tree Nodes .....	17
Adding Options to the New Menu .....	18
Adding Items To Menus .....	19
Static Menu Items .....	20
Internal Frame Menu Items .....	21
Console Tree Menu Items .....	21
Handling Save As .....	21
Handling Server Connection and Disconnection .....	22
Standard Controls .....	23

The StandardDialog Class .....	23
Name of Standard Dialog Class .....	24
Dialog Creation .....	24
Dialog Initialization .....	24
Dialog Default Action .....	25
Dialog Keyboard Handling, Focus Order, Action Maps, and So On .....	25
Dialog Results .....	26
Methods to Override .....	26
Standard Buttons and Other Controls .....	27
Administration Services Console Services .....	28
Retrieving the CSS Token from the Console .....	28
Sending E-mail .....	28
Internationalization .....	29
Packaging the Plug-in .....	29
<b>Chapter 3. Writing Server-side Command Listeners .....</b>	<b>31</b>
Prerequisites .....	31
Command Listeners .....	32
Class Hierarchy .....	32
Which Class To Extend .....	33
Which Methods to Override .....	33
Registering Commands .....	34
CommandString Class .....	34
CommandArgument Class .....	34
CommandDescriptor Class .....	35
Examples .....	35
Command Handling Methods .....	38
Method Signatures .....	39
Grabbing Command Arguments .....	40
Sending Results Back to the Client .....	40
Storing Temporary Data Using the Framework .....	41
Packaging the Code .....	42
Loading the Code .....	43
Utility Classes .....	44
<b>Glossary .....</b>	<b>45</b>
<b>Index .....</b>	<b>57</b>

---

# Documentation Accessibility

---

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at <http://www.oracle.com/accessibility/>.

## Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

## Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

## Access to Oracle Support for Hearing-Impaired Customers

Oracle customers have access to electronic support through My Oracle Support or by calling Oracle Support at 1.800.223.1711. Hearing-impaired customers in the U.S. who wish to speak to an Oracle Support representative may use a telecommunications relay service (TRS). Information about the TRS is available at <http://www.fcc.gov/cgb/consumerfacts/trs.html/>, and a list of telephone numbers is available at <http://www.fcc.gov/cgb/dro/trsphonebk.html>. International hearing-impaired customers should use the TRS at +1.605.224.1837. An Oracle Support engineer will respond to technical issues according to the standard service request process.



---

## In This Chapter

About Administration Services.....	7
About Java Plug-in Components.....	8
Requirements for Using Administration Services Java Plug-ins.....	9
Prerequisite Knowledge .....	9
Framework Concepts .....	10
About the Sample Code in this Guide .....	10

This chapter provides an overview of Oracle Essbase Administration Services.

## About Administration Services

Administration Services is the cross-platform framework for managing and maintaining Oracle Essbase. Administration Services provides a single point of access for viewing, managing, and maintaining Essbase Servers, Essbase Administration Servers, and Oracle Hyperion Provider Services.

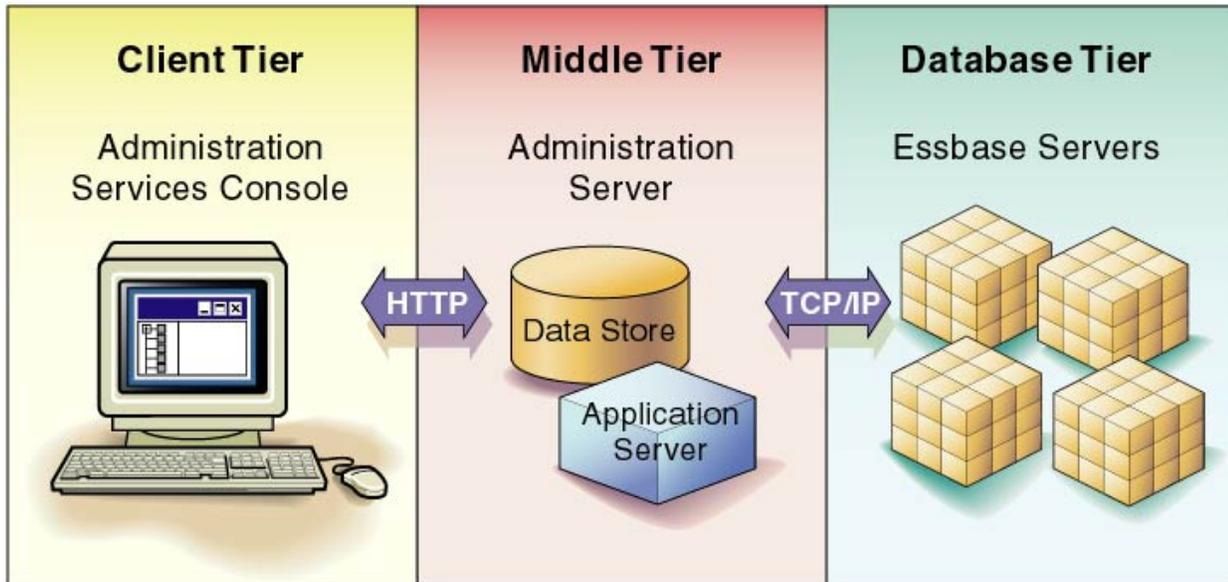
Administration Services works with Essbase Servers in a three-tiered system that consists of a client user interface, a middle-tier server, and one or more database servers (Essbase Servers). The middle tier coordinates interactions and resources between the user interface and Essbase Servers. The three tiers may or may not be on the same computer or platform. For more information about deployment scenarios, see *Oracle Hyperion Enterprise Performance Management System Installation and Configuration Guide*.

The three tiers include the following:

- Client tier: Administration Services Console—A Java client console enabling administrators to manage the Essbase environment from a robust graphical user interface.
- Middle tier: Essbase Administration Server—A Java middle-tier server that communicates with Administration Services Console and Essbase Servers. Essbase Administration Server maintains communication and session information for each connection to Essbase Servers. Essbase Administration Server also stores documentation files so that console users can access documentation without having to install it locally.
- Database tier: Essbase Server—One or more Essbase Server that store and process multidimensional database information. Essbase Servers are installed separately from Administration Services.

Essbase Administration Server serves as the middle tier between Administration Services Console and Essbase Servers, as shown in [Figure 1](#).

Figure 1 Administration Services Architecture



## About Java Plug-in Components

Administration Services Java plug-ins are installable components. They provide the following benefits to users:

- Enable the Administration Services development team to easily provide additional functionality to end users
- Allow other Oracle internal development groups to easily integrate their products with Administration Services
- Enable partners and customers to easily integrate their processes into Administration Services
- Allow customers to accomplish more because they are not launching several applications at once

The following list describes how you can use Administration Services plug-ins:

- Customize the Administration Services Console Enterprise Tree
- Customize the Administration Services Console File > Open dialog box
- Customize the Administration Services Console File > New dialog box
- Customize the Administration Services Console File > Save As dialog box
- Change the Administration Services Console menus

For each of these tasks, there are a set of classes, interfaces, and methods that must be implemented by a plug-in author. There are also a set of guidelines to follow when implementing plug-ins.

For information about performing the preceding tasks, see [“Writing Client Plug-ins” on page 11](#).

## Requirements for Using Administration Services Java Plug-ins

The following list describes the requirements necessary to use Administration Services Java plug-in components:

- Java SDK Version 1.4.1\_b06 or later
- Essbase Release 7.1 or later
- Administration Services Release 7.1 or later

## Prerequisite Knowledge

Developers using this guide must have the following prerequisite knowledge:

- XML (Extensible Markup Language)
- HTTP (Hypertext Transfer Protocol)
- Java 2 Introspection
  - Introspection is a Java technique that Administration Services uses to interact and communicate with plug-in components.
  - Exception handling
  - Packaging of applications (.jar files)
- Swing

Swing is a graphical user interface (GUI) component kit, part of the Java Foundation Classes (JFC) integrated into Java 2 platform, Standard Edition (J2SE). Swing simplifies deployment of applications by providing a complete set of user-interface elements written entirely in the Java programming language. Swing components permit a customizable look and feel without relying on any specific windowing system.

Because Swing is incorporated in the Java 2 platform, there is no need to download or install it.

# Framework Concepts

## Packaged APIs for Administration Services

Administration Services consists of several packages. For detailed information about these packages, see the *Administration Services Java API Reference* for the packages and classes described in this guide.

### Administration Services Java Packages

com.essbase.eas.ui.\* (all packages)

com.essbase.eas.framework.\* (all packages)

### Example Classes

ConsoleTreeHandler

ConsoleMenuHandler

MiscellaneousHandler

NewDialogHandler

OpenDialogHandler

OptionsDialogHandle

## About the Sample Code in this Guide

The code snippets and examples contained in this guide are intended to demonstrate how plug-ins interact with the Administration Services framework. They are intended to show how to get an aspect of the interaction to work and, in some cases, omit details that are not relevant to the topic being discussed. In addition, while the techniques shown will work, the Java techniques shown may in some cases not be the best implementation method when scaling up to a production quality product.

For example, in the section on context menu items, [“Adding Context Menu Items To Tree Nodes” on page 17](#), the example creates new menu items and action listeners each time the `getContextMenuItems()` method is called; this might not be the best mechanism for handling this task. Please consult the appropriate Java resources (books, Web pages, documentation) for other techniques; in particular, when dealing with Swing objects, the Swing event model, and associating Swing event listeners to objects.

# 2

## Writing Client Plug-ins

### In This Chapter

Access Point for Plug-ins .....	11
Class Packages .....	11
How the Client Locates Plug-ins .....	13
Creating the Miscellaneous Handler Class .....	13
Adding Functionality .....	14
Standard Controls .....	23
Administration Services Console Services .....	28
Internationalization .....	29
Packaging the Plug-in .....	29

This chapter explains how to write a plug-in for Administration Services Console. Plug-ins are the mechanism for extending the functionality of Administration Services Console.

### Access Point for Plug-ins

The implementation of the Administration Services client is contained in the `eas_client.jar` and `framework_client.jar` files that are installed with Administration Services. Additional classes are found in the `eas_common.jar` and `framework_common.jar` files. The Essbase plug-in to Administration Services Console is contained in the `essbase_common.jar` and `essbase_client.jar` files.

### Class Packages

Administration Services Console consists of several packages. The public classes in these packages are available to the implementor of plug-ins. In particular, the user interface, print, and mail-related classes. For detailed information about the packages and classes described in [Table 1](#), see the *Administration Services Java API Reference*.

**Table 1** Administration Services Console Class Packages

Package or Class Name	Description
<code>com.essbase.eas.client.intf</code>	The classes and interfaces that provide an interface to the console

<b>Package or Class Name</b>	<b>Description</b>
com.essbase.eas.client.manager	The classes that provide “management” services for parts of the console; such as, LoginManager, CommandManager, ConsoleManager, and so on
com.essbase.eas.client.plugins	The classes that the client framework uses to install plug-ins, track plug-ins, and so on
com.essbase.eas.framework.client.defs.command	The client-specific classes related to sending commands to the mid-tier. As of Release 7.1, this consists only of the UICommandManager class.
com.essbase.eas.framework.client.defs.login	This is the default login dialog box provided by the console. It displays if no plug-in has registered a different login dialog or if any command is sent to the Administration Services mid-tier and a mid-tier server name has not been provided.
com.essbase.eas.framework.client.ui.fileDlgs	Implements dialog boxes associated with a file menu. For example, New, Open, Save As
com.essbase.eas.ui	Another package with several user interface components used by the console and by the Essbase plug-in
com.essbase.eas.ui.ctable	An implementation of a standard extension to the JTable control
com.essbase.eas.ui.ctree	An implementation of an extension to the JTree control. This is the control that is used in the Enterprise Tree and in the custom views of the console.
com.essbase.eas.ui.editor	An implementation of a standard text editor with syntax highlighting. This control is used as the base class for the calculation script editor, MaxL editor, and report script editor in the Essbase plug-in.
com.essbase.eas.ui.email	An implementation of some e-mail related classes. The framework provides a service for sending e-mail; this package contains the implementation of the service.
com.essbase.eas.ui.font	The classes that provide the font-related utility
com.essbase.eas.ui.print	The classes that provide the print-related utility
com.essbase.eas.ui.phtable	An extension to the JTable control for editing properties. This table provides extensive editing, sorting capabilities, and is used by many windows and dialogs in the Essbase plug-in.
com.essbase.eas.ui.ptree	An extension to the JTree control for editing tree-oriented properties. This tree provides extensive editing capabilities and is used by many windows and dialogs in the Essbase plug-in.
com.essbase.eas.ui.tree	The generic utility routines for working with JTree-based controls
com.essbase.eas.framework.defs	This package and the packages under it provide services for transferring commands from the mid-tier to the client, packaging/unpackaging data to be transferred, a logging mechanism, and so on
com.essbase.eas.i18n	The internationalization utility classes
com.essbase.eas.utils	Various utility classes spanning a range of uses: file utilities, compression, encryption, array utilities, and so on
com.essbase.eas.utils.print	Utility classes dealing with printing

## How the Client Locates Plug-ins

The client tracks plug-ins by maintaining a list of jar files that the user has selected using the Configure Plugin Components dialog box. To display this dialog box, from Administration Services Console, select **Tools**, and then **Configure components**.

When a jar file is selected, the dialog scans through each package in the jar file looking for a class called `MiscellaneousHandler.class`. When a class with this name is found, the jar file name and the package name containing that class file are retained by the plug-in manager. Therefore, each jar file must contain exactly one package with a `MiscellaneousHandler` class in it.

When Administration Services Console starts, the plug-in manager scans each jar file in its stored list, looking for the `MiscellaneousHandler.class` file in the specified package. If this class is found, the plug-in manager adds this plug-in to its list of plug-ins. Other parts of the application, or any other plug-in can then call the plug-in manager to get a list of all plug-ins.

Basically, each plug-in consists of the following:

- A jar file containing a package with a  
    `MiscellaneousHandler` class

For the rest of this document, we will use the term “plug-in root” to refer to the package containing the `MiscellaneousHandler` class.

For example, the rest of this document uses a plug-in with a class named `com.MyPlugin.MiscellaneousHandler`; the plug-in root refers to the package `com.MyPlugin`.

## Creating the Miscellaneous Handler Class

In order for Administration Services to recognize your client plugin, you must create a `MiscellaneousHandler.java` class and include it in the plugin jar file. See the following example, which implements a single API `getDescription()`. The `framework_client.jar` file is required to compile this example.

```
package com.mycompany.client.plugins;

import com.essbase.eas.client.plugins.Description;

public class MiscellaneousHandler {
    public MiscellaneousHandler() {
    }
    /**
     * Return a description object for this plugin
     */
    public static Object getDescription() {
        Description d = new Description();
        d.setText("Give a short description");
        d.setVersion("1.0.0");
        d.setVendor("My Company Inc.");
        d.setCopyright("Copyright 2006, My Company Inc.");
        return d;
    }
}
```

## Adding Functionality

There are many ways to add functionality to Administration Services Console. The following sections describe how this is currently implemented:

- [“Semantic Rules” on page 14](#)
- [“Adding a Branch to the Enterprise Tree” on page 14](#)
- [“Adding Children to Other Tree Nodes” on page 16](#)
- [“Permitting Plug-ins To Add Children To Your Tree Nodes” on page 17](#)
- [“Adding Context Menu Items To Tree Nodes” on page 17](#)
- [“Adding Options to the New Menu” on page 18](#)
- [“Adding Items To Menus” on page 19](#)
- [“Handling Save As” on page 21](#)
- [“Handling Server Connection and Disconnection” on page 22](#)

## Semantic Rules

Many of the following sections have a description of semantic rules. In most cases, Administration Services Console does not enforce these rules. We expect that developers writing plug-ins for Administration Services will be “well-behaved citizens”; philosophically, this means that a lot of the console is open, accessible, and plug-ins can have an adverse effect on the application by taking actions that break these semantic rules.

## Adding a Branch to the Enterprise Tree

When Administration Services Console starts, a panel is created called the “Enterprise View”. This panel contains an instance of the CTree class. The text for the root node is called “Enterprise View”. Each plug-in gets the opportunity to add children to the root node. This permits each plug-in to have its own branch in the Enterprise Tree view.

In the plug-in root, add a class called ConsoleTreeHandler. In our example, this would be `com.MyPlugin.ConsoleTreeHandler`. Add a method called “`populateTree()`” to this class. The source code should look something like the following example:

```
public class ConsoleTreeHandler {
    //a no-argument constructor is required by the framework.
    public ConsoleTreeHandler() {
    }

    public void populateTree(CTreeModel model) {
        Object root=model.getRoot();

        //strictly speaking, this next check should not be
        //necessary; however, we do this to make sure some other
        //plug-in hasn't replaced the root node with something
        //unexpected.
        if ((root!=null) && (root instanceof CTreeNode))
```

```

        //create any CTreeNode-derived objects, adding them
        //as children of the root node.
    }
}
}

```

There are some unenforced semantic rules associated with CTree objects:

- The only action a plug-in should perform on the CTreeModel is to get the root. The plug-in should never replace the root node, traverse the tree model, or make changes to any other descendants of the root node.
- Every object added as child of the root node must be derived from a CTreeNode. Theoretically, any object can be added as a child of the root; however, other parts of the framework will not respond to those objects in any meaningful way.

**Note:** A plug-in can be called more than once if the console disconnects from the current server. The code needs to check that the node has already been added and only append nodes that have not been added previously. The source code should look something like the following Essbase ConsoleTreeHandler code:

```

/**
 * populates the model with information required.
 */
public void populateTree(CTreeModel model) {
    Object root=model.getRoot();
    CTreeNode rootNode=null;
    boolean firstTime=true;
    if (root instanceof CTreeNode) {
        rootNode=(CTreeNode) root;
        if (rootNode.getChildCount()!=0) {
            CTreeNode node=(CTreeNode) rootNode.getFirstChild();
            while (node !=null) {
                if (node instanceof ServersContainerNode) {
                    firstTime=false;
                    UIFactory.refreshServerList();
                    break;
                }
                node=(CTreeNode) rootNode.getChildAfter(node);
            }
        }
    }
    if (firstTime) {
        CTreeNode essnode=new ServersContainerNode(null);
        rootNode.add(essnode);
        final CTreeNode containerNode=essnode;

        ConsoleManager.getConsoleInstance().addFrameListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                //signal that we are simply disconnecting instead of
                //closing
                if (e.getNewState() == WindowEvent.WINDOW_OPENED &&
                    e.getOldState() == WindowEvent.WINDOW_OPENED) {
                    Server[] servers = UIFactory.getServers();
                    for (int ii=0; ii<servers.length; ii++) {

```

```

        UIFactory.removeServerInstance(servers[ii]);
    }
}
UIFactory.disconnectAll();
}
}))
}
}

```

## Adding Children to Other Tree Nodes

When a `CTreeNode` object is expanded for the first time, each plug-in gets the opportunity to add child nodes to the `CTreeNode` being expanded.

In the plug-in root, add a class called `ConsoleTreeHandler`. In our example, this would be `com.MyPlugin.ConsoleTreeHandler`. Add a method called “`getTreeNodeChildren()`” to this class. The source code should look something like the following example:

```

public static CTreeNode[] getTreeNodeChildren(CTreeNode node) {
    // strictly speaking, this check for null should never be
    // necessary
    if (node == null)
        return new CTreeNode[0];
    if (node instanceof SomeSpecificTreeNode) {
        CTreeNode[] theChildren = new CTreeNode[5];
        theChildren[0] = new ChildNode();
        theChildren[1] = new AnotherChildNode();
        // and so on...
        return theChildren;
    }
    else if (node instanceof SomeOtherTreeNode) {
        // different set of children here.
    }
    // and if we're not interested in any other types.
    return new CTreeNode[0].
}
}

```

Item of interest for this operation:

- This method could be declared `public Object[] getTreeNodeChildren(CTreeNode node)` and it would still get called. The `CTreeNode` method that handles this checks the return value for null and also checks each item returned in the array to ensure that it is an instance of a `CTreeNode` object. Declaring the method as in the example enforces to the implementer of the plug-in that the items returned must be items derived from the `CTreeNode` class.
- The only arrangement that currently is done is that child nodes that cannot have children are placed before the child nodes that can have children. Nodes from plug-ins are placed after the nodes that the parent node already knows about.

## Permitting Plug-ins To Add Children To Your Tree Nodes

By default, all CTreeNode based objects that can have children have this feature enabled. Currently, there is no way to prevent plug-ins from adding children to a tree node if that tree node can have children.

## Adding Context Menu Items To Tree Nodes

When the CTree control detects that a popup menu needs to be displayed, it calls the instance of the CTreeNode and asks it for a list of items to display in the context menu. The following are rules or guidelines for how CTreeNode objects should build this array:

- The signature for the CTreeNode method is:

```
public Component[] getContextMenuItems();
```

Even though this method is declared to return an array of Component objects, it is highly recommended that the objects returned all be instances of the JMenuItem class (or classes derived from JMenuItem).

- The state of any menu items returned from the getContextMenuItems() method must be properly initialized; that is, enabled/disabled, checked.
- The JMenuItem objects (or whatever objects) must be properly linked to the specific CTreeNode object that is being called. The event passed in the actionPerformed() call will contain none of this contextual information.

The CTree then calls each plug-in, retrieving any additional menu items for the specified CTreeNode object. If there are additional items, the CTree places a separator after the original menu items, then places all of the plug-in items in the popup menu, and then, if the CTreeNode can be put on custom views, puts another separator and the menu items related to custom views.

For a plug-in to respond to the CTree properly in this case, add a class called ConsoleTreeHandler to the plug-in root package. In our example, this would be com.MyPlugin.ConsoleTreeHandler. Add a method called “getContextMenuItemsFor()” to this class. The source code could look something like the following example:

```
public static Component[] getContextMenuItemsFor(CTreeNode node) {
    // strictly speaking, this check for null should never be
    // necessary
    if (node == null)
        return new Component[0];
    if (node instanceof SomeSpecificTreeNode) {
        JMenuItem theItem = new JMenuItem("Walk");
        JMenuItem anotherItem = new JMenuItem("Don't walk");
        theItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // take action here.
            }
        })
        return new Component[] { theItem, anotherItem };
    }
    else if (node instanceof SomeOtherTreeNode) {
        // different set of menu items here.
    }
}
```

```

}
// and if we're not interested in any other types.
return new Component[0].
}

```

Items of interest for this operation:

- This method can be declared to return anything. For instance, for better type safety within your own code, you could declare the method to be “public static JMenuItem[] getContextMenuItemsFor(CTreeNode node)”; however, the CTree object making the call will only use items that are derived from the Component class.
- This example is very bare bones; for instance, the returned JMenuItem object does not know which CTreeNode object it should be working with; even worse, one of the items does not have an action listener associated with it. For a complete example of this, please see the sample plug-ins developed by the Administration Services development team.
- CTreeNode (being derived from DefaultMutableTreeNode) objects have a user object. This is available through the getUserObject() method. The intent is that the user object for a node represents that data that the node has been created for and this is the data that would need to be associated with the menu item. For instance, a node might have an object representing an Essbase application. In the above example, we would then perform a node.getUserObject() call to obtain this Essbase application object
- Because plug-ins are called in the order that the user has arranged them in the Component Manager dialog box, there currently is no way to force the menu items from one plug-in to appear before the menu items of another plug-in.

## Adding Options to the New Menu

The Administration Services development framework provides one dialog box for the New menu. When New is selected, the framework creates and displays an instance of the `com.essbase.eas.framework.client.ui.filedlg.NewDialog.java` class. The Essbase plug-in and Administration Services plug-in add the following tabs:

- Essbase
- Scripts
- Wizards

The dialog box class includes the following items:

- The OK, Cancel, and Help buttons
- An instance of a JTabbedPane to act as a container for each of the other panels
- Actions for the OK, Cancel, and Help buttons that make the appropriate calls into the plug-in that provided the active panel

To add a panel and tab to the New dialog box, add a class called `NewDialogHandler` to the plug-in root package. In our example, this would be `com.MyPlugin.ConsoleTreeHandler`. Add a method called “`populatePanel()`” to this class. Example source code:

```
public void populatePanel(JTabbedPane panel) {
```

```

// create an instance of the right kind of panel
CNewDialogScrollPane s = new CNewDialogScrollPane();
s.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
s.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

// create a list model that has some items in it.
DefaultListModel model = new DefaultListModel();
model.addElement(new JLabel("XTD Connection"));
model.addElement(new JLabel("SQL Connection"));

// make sure the list box has a selected item
list.setSelectedIndex(0);

// toss the list into the scroll pane and ensure that the new
// dialog box will call this instance when the OK button is
// clicked.
s.getViewPort().add(list);
s.setOkHandler(this);

// add this panel to the tabbed panel we were given
panel.add("My Objects", s);
}

```

For this to work correctly, you must add the following method to the class:

```

public void handleOk(Component component) {
    if (component instanceof CNewDialogScrollPane) {
        CNewDialogScrollPane scroller = (CNewDialogScrollPane) component;
        Component control = scroller.getViewPort().getComponent(0);
        if (control != null) && (control instanceof JList) {
            // extract the selected item in the JList.
            // ensure that it is one of the ones we added.
            // take the appropriate action.
        }
    }
}

```

Items of interest for this operation:

- Items added to the JTabbedPane must be derived from the CNewDialogScrollPane class.
- Because CNewDialogScrollPane is derived from JScrollPane, components derived from JTable, JTree, and JList display best in the New dialog box.
- Components added to the scroller can have custom renderers, event handlers, and so on.
- For the best behavior, this list would need a MouseListener added to listen for double-click events. This MouseListener should then call the enclosing dialog box's handleOk() method.
- A plug-in can add more than one panel to the JTabbedPane instance.

## Adding Items To Menus

Menu items are typically displayed in three ways:

- Static

- From an internal frame
- From a CTreeNode on the console tree

## Static Menu Items

Static menu items are always displayed. The following example is for a static menu item:

```
public class XYZ {
    private CMenu editorsMenu = new CMenu("Scripts", Console.ID_ACTIONS_MENU - 1, this);
    private CMenuItem outline = new CMenuItem("Outline", null, 0, this);
    private CMenuItem report = new CMenuItem("Report", null, 1, this);
    private CMenuItem calc = new CMenuItem("Calc", null, 2, this);
    private CMenuItem maxl = new CMenuItem("Maxl", null, 3, this);
    private CMenuItem mdx = new CMenuItem("Mdx", null, 4, this);
    private CMenuItem dataprep = new CMenuItem("DataPrep", null, 5, this);
    void createMenu() {
report.addActionListener(new AbstractAction("createReport") {
    public void actionPerformed(ActionEvent e) {
    }
});
calc.addActionListener(new AbstractAction("createCalc") {
    public void actionPerformed(ActionEvent e) {
    }
});
maxl.addActionListener(new AbstractAction("createMaxl") {
    public void actionPerformed(ActionEvent e) {
    }
});
mdx.addActionListener(new AbstractAction("createMdx") {
    public void actionPerformed(ActionEvent e) {
    }
});
outline.addActionListener(new AbstractAction("createOutline") {
    public void actionPerformed(ActionEvent e) {
    }
});
dataprep.addActionListener(new AbstractAction("createDataPrep") {
    public void actionPerformed(ActionEvent e) {
    }
});
editorsMenu.add(outline);
editorsMenu.add(dataprep);
editorsMenu.add(calc);
editorsMenu.add(report);
editorsMenu.add(maxl);
editorsMenu.add(mdx);
LocalizeUtils.localizeMenu(resources, editorsMenu);
ConsoleManager.getConsoleInstance().mergeMenus(new Component[] { editorsMenu});
    }
}
```

## Internal Frame Menu Items

Menu items from an internal frame only display when the internal frame is active. If the internal frame is deactivated or closed, then these menu items no longer are displayed. The following example is for an internal frame menu item:

```
public class XYZ extends CInternalFrame {
    public Component[] getFrameMenus() {
// Like the example above
        return (new Component[] { editorsMenu});
    }
}
```

## Console Tree Menu Items

These menu items only display when a node is selected. The following example is for a console tree menu item:

```
public XYZ extends CTreeNode {
    public Component[] getActionMenuItems() {
        return (new Component[] { editorsMenu});
    }
}
```

In general, there are predefined menu positions defined in the Console interface:

```
public static final int ID_FILE_MENU = 0;
public static final int ID_EDIT_MENU = 1;
public static final int ID_VIEW_MENU = 2;
public static final int ID_ACTIONS_MENU = 10;
public static final int ID_TOOLS_MENU = 20;
public static final int ID_WIZARD_MENU = 30;
public static final int ID_WINDOW_MENU = 90;
public static final int ID_HELP_MENU = 99;
```

If the CMenu item's (that is returned from the above example) position matches with one of the predefined ones, then that CMenu item's submenus are merged in else that CMenu is inserted based on the position. So if the CMenu has a position of ID\_ACTIONS\_MENU, then the items are merged in to the action menu item that is already on the main menubar. If the CMenu has a position (ID\_ACTIONS\_MENU - 1), then the CMenu is inserted before the action menu.

## Handling Save As

Save As requires the plug-in to implement the interface SaveAsRequestor. The following example uses an inner class:

```
if (saveAsAdapter == null) {
    saveAsAdapter = new SaveAsAdapter();
}
SaveAsDialog.showDialog(resources.getString("exportTitle"), (SaveAsRequestor)
saveAsAdapter);
}
```

The `initSaveAsDialog` is called to allow the dialog/frame to initialize the `SaveAsDialog` as it needs to. By default a file system chooser is added to `mainPanel` at index 0. A plug-in can add other panels to save to other places in this method.

When an object is selected from any panel, then the `saveAsObject` method is called with the selected object. If the file system panel is selected the object will be a `File` if the plug-in adds a panel of their own it they will have to perform the steps to save the object.

```
private class SaveAsAdapter implements SaveAsRequestor {
    public void initSaveAsDialogComponents(JTabbedPane mainPanel) {
        String xmlString = ResourceUtilities.getStringSafely(resources, XML_FILES);
        DefaultFileFilter xmlFilter = new DefaultFileFilter(xmlString, "xml", resultAction);
        JFileChooser jfc = (JFileChooser) mainPanel.getComponentAt(0);
        jfc.setSelectionMode(JFileChooser.FILES_ONLY);
        if (jfc.isAcceptAllFileFilterUsed() == true)
            jfc.setAcceptAllFileFilterUsed(false);
        jfc.setFileFilter(xmlFilter);
    }

    public void initExtraComponents(JPanel extraPanel) {
    }

    public boolean saveAsObject(Object saveObject) {
        boolean saved = false;
        if (saveObject instanceof File) {
            File file = (File) saveObject;
            String exportFile = file.getPath();
            if (exportFile != null) {
                String msg = "";
                if (AdminServerPropertiesHelper.requestExportDB(exportFile))
                {
                    msg = resources.getString("sucEXDBMsg");
                    StandardMessages.showMessageDialog(resources, "exportTitle", msg,
                        JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE);
                    saved = true;
                }
                else
                {
                    msg = resources.getString("faileXDBMsg");
                    StandardMessages.showMessageDialog(resources, "exportTitle", msg,
                        JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
                }
            }
        }
        return saved;
    }
    public void setFocusComponent() {
    }
}
```

## Handling Server Connection and Disconnection

In this release, Administration Services Console initially opens up disconnected from any Essbase Administration Server. Your code can be notified when Essbase Administration Server is connected, is disconnecting, or has already disconnected. Your code can implement the

EASServerListener interface to be notified of the change in the Essbase Administration Server state.

```
import com.essbase.eas.client.intf.EASServerListener;
private EASServerListener listener = new EASServerListener() {
    public void ServerDisconnecting(String server) {
        // server is about to disconnect.
    }
    public void ServerDisconnected(String name) {
        // server is disconnected
        // disable menu items or Enterprise tree nodes when disconnected
        // from the EAS server
    }
    public void ServerConnected(String name) {
        // enable menus
        // add Enterprise tree nodes
    }
};
```

To add the EASServerListener to the console use the following code snippet.

```
Console console = ConsoleManager.getConsoleInstance();
console.addEASServerListener(listener);
```

## Standard Controls

While it is not required that plug-ins use the standard controls provided by the framework classes, there are some benefits to using them. Namely, some consistency of look and feel is provided, some housekeeping tasks are performed by the standard controls, there is support for internationalization, accessibility, and so on.

## The StandardDialog Class

The StandardDialog class is an extension of the JDialog class and was introduced for the following reasons:

1. Standardize the mechanism for internationalization and localization handling
2. Standardize the position, location, and behavior of dialog “action” buttons
3. Standardize some of the accessibility handling for modal dialogs
4. Standardize the handling of results

The StandardDialog class contains the following protected (or private) fields:

**Table 2** Fields in the StandardDialog Class

Field	Description
okBtn	An instance of an OK button. This is one of the standard controls described in <a href="#">“Dialog Initialization” on page 24</a> .

Field	Description
cancelBtn	An instance of a Cancel button. This is one of the standard controls described in <a href="#">“Dialog Initialization” on page 24</a> .
helpBtn	An instance of a Help button. This is one of the standard controls described in <a href="#">“Dialog Initialization” on page 24</a> .
buttons	An instance of a ButtonPanel. The ButtonPanel is one of the standard controls described in <a href="#">“Dialog Initialization” on page 24</a> .
resources	An instance of a ResourceBundle object. This resource bundle is used for internationalization purposes.
adapter	An instance of a StandardDialogAdapter.
dialogResult	An instance of a DialogResult object.
saveDialogBounds	A boolean value indicating whether the bounds (location and size) of this dialog should be saved when it is closed.

## Name of Standard Dialog Class

The name of the Standard Dialog class is StandardDialog. It is in `com.essbase.easui.StandardDialog.class`.

## Dialog Creation

There are at least 11 constructors for the StandardDialog class; most of these chain to another constructor. The two constructors that should be invoked by derived classes are the ones with the following signatures:

- `StandardDialog(Frame owner, String title, boolean modal, DialogResult result);`
- `StandardDialog(Dialog owner, String title, boolean modal);`

Most of the other constructors exist only to match constructor names of the JDialog class.

## Dialog Initialization

During the call to the StandardDialog constructor, the following initialization steps will occur:

- An OK button, a Cancel button, and a Help button are created
 

These are the standard buttons used by most dialogs. If the dialog being implemented uses a different set of buttons (for instance, Close, Apply, Next, and so on) the derived class should implement instances of those buttons.
- A ButtonPanel containing the OK, Cancel, and Help buttons is created
 

If the dialog being implemented wants the button panel to contain a different set of buttons, it should call `buttonPanel.changeButtons(new JButton[] { closeBtn, helpBtn });` // as an example.
- A ResourceBundle instance is created

This resource bundle is used to perform localization work within the dialog. It is important to know where the standard dialog looks for the instance of the resource bundle. For example, if the dialog class is `MyFunnyDialog`, then the resource bundle must be in a file called `resources/MyFunnyDialog.properties`.

- A `StandardDialogAdapter` is created and is added as a window listener to the dialog

---

**Caution!** Because of the implementation of the `StandardDialogAdapter` class, there should never be a reason for a descendant class of `StandardDialog` to attach a `WindowListener` to itself. Routing of all window events should be handled by the `StandardDialogAdapter`. If the descendant class needs to take action when a window close, window open, and so on, event occurs then override the methods in `StandardDialog` that the `StandardDialogAdapter` calls.

---

- Sets the instance of the dialog result to the value passed in, if any  
To understand how this works, see [“Dialog Results” on page 26](#).
- Sets the dialog’s default close operation to `DISPOSE_ON_CLOSE`  
In most cases, this is the desired behavior; for a dialog that needs a different behavior, this can be changed by the constructor in the descendant class.
- Sets the dialog’s content pane layout to be a `BoxLayout` oriented vertically  
If necessary, this can be changed by the derived class.
- Adds entries to the action and input maps of the dialog’s root pane to take a “default action” when the Enter key is pressed by the user  
For more information on what this default action is, and why this step is necessary, see the section of this document titled “Dialog default action”.

## Dialog Default Action

The Microsoft Windows operating environment has the concept of a default button when modal dialog windows are open. The default button is painted in a way that makes it stand out visually to the user. Normally, that is the OK button; however, it can be any action button on the dialog. To handle this concept, the `StandardDialog` adds entries to the action and input maps of its root pane for handling the enter keystroke.

If your dialog box does not have an OK button or, if at any time, the default button should be some other button, then a call like the following needs to be performed:

```
dlg.getRootPane().setDefaultButton(closeBtn);
```

## Dialog Keyboard Handling, Focus Order, Action Maps, and So On

Depending on which buttons are inserted into a dialog, certain keystrokes will be mapped automatically:

- The Enter key
- The Esc key

- The F1 key (for help)

These are the primary keystrokes that are mapped by the standard dialog and the standard buttons.

To add handling when these keystrokes are pressed, do the following:

- For the Enter key, override the `handleOk()` method. If everything finishes correctly and the dialog needs to be released, then call `super.handleOk()`. This will ensure that the dialog shuts down properly.
- For the Esc key, override the `handleCancel()` method. The standard dialog behavior closes the dialog, releases all the controls, disposes of contained components, and so on. In most cases, this method will not need to be overridden.
- For the F1 key, override the `handleHelp()` method. If the dialog has been connected via the Administration Services help system via the normal manner, this step should not be necessary.

By default, the Java Swing implementation sets the focus order of controls to correspond to the order in which they were added to their container, and then those container's to their container, and so on. This can be overridden by making a call to the method `DialogUtils.setFocusOrder()`. This mechanism should be used in all dialogs to ensure the focus order of controls is correct and doesn't rely on how the code for building the containment models was written.

## Dialog Results

In many cases, a dialog needs to return a significant amount of information to the calling mechanism. Unfortunately, the method `Dialog.show()` is declared as void and does not return any data.

If, when implementing a dialog, results from the dialog are needed, the recommended way to get those is by doing the following tasks:

- Extend the `DialogResult` class to contain references and additional data needed by the dialog and/or returned by the dialog.
- Before creating the dialog, create an instance of the `DialogResult` class.
- Ensure that the dialog has at least one constructor that accepts an instance of a `DialogResult` object.
- In the constructor for the dialog class derived from `StandardDialog`, pass the `DialogResult` object to the correct `StandardDialog` constructor.
- During the handling of the OK button, set the results back into this instance.

## Methods to Override

The `StandardDialog` class has a set of methods that can be overridden. Whether each of these methods are overridden will depend on the needs of each derived class. See the *Administration Services Java API Reference* for detailed information about each of the following methods:

- dispose()
- handleCancel()
- handleOk()
- handleWindowClosed()
- handleWindowClosing()
- handleWindowOpened()

## Standard Buttons and Other Controls

There are a large number of standard controls provided by the client framework. The following is a representative list; for more complete information, see the *Administration Services Java API Reference* for the com.essbase.eas.ui package and descendant packages.

**Note:** This is not a complete list of controls. The plug-in developer should browse the Java API Reference for the com.essbase.eas.ui package and other packages under this one for additional standard components.

- ActivateButton
- ApplyButton
- BackButton
- BooleanComboBox
- ButtonPanel
- CancelButton
- CloseButton
- DoneButton
- FinishButton
- HelpButton
- ListMoverPanel
- NextButton
- NumericTextField
- OkButton
- ReadOnlyTextFrame
- RefreshButton
- ResetButton
- SimpleWizardPanel
- VerticalPairPanel
- WizardPanel

# Administration Services Console Services

The client framework provides the following Administration Services Console services:

- [Retrieving the CSS Token from the Console](#)
- [Sending E-mail](#)

## Retrieving the CSS Token from the Console

The CSS token is retrieved from the FrameworkUser object which is returned on successful login to Essbase Administration Server.

```
import com.essbase.eas.client.intf.Login;
import com.essbase.eas.client.manager.LoginManager;
import com.essbase.eas.admin.defs.*;
import com.essbase.eas.admin.client.*;
import com.essbase.eas.framework.defs.FrameworkUser;
private String getToken() {
    String loginToken = null;
    Login login = LoginManager.getLoginInstance();
    if (login != null) {
        FrameworkUser u = (FrameworkUser)
login.getProperty("FrameworkUser");
        if (u != null) {
            loginToken = u.getToken();
        }
    }
    return loginToken;
}
```

## Sending E-mail

Administration Services Console has integrated support for sending e-mail using the JavaMail API. We have wrapped the classes and provide a dialog for sending e-mail. There is also support in the InternalFrame class to send from any class derived from the CInternalFrame class.

The following is a simple example of how to send the contents of a text area in an e-mail from a dialog.

```
Import com.essbase.eas.ui.email.*;
public void email() {
    JFrame fr = ConsoleManager.getConsoleFrame();

    SendEmail email = new SendEmail(fr, fr.getTitle(), new Object[] {
getTextArea().getText() } );
    email.send();
}
```

The following example is for a window derived from CInternalFrame. The methods, isEmailable() and getObjectsToEmail, are methods in the CInternalFrame class.

```
public boolean isEmailable() {
    return true;
}
```

```

}

public Object[] getObjectsToEmail() {
    HTMLDoc doc = new HTMLDoc();

    doc.setTitle(getTitle());
    doc.addObject(doc.getHeading(2, doc.getStyleText(getTitle(),
doc.BOLD | doc.UNDERLINE), doc.CENTER));
    doc.addObject(doc.BR);
doc.addObject(TableUtilities.getHTML((DefaultTableModel) locksTable.getModel()));
    return (new Object[] { new EmailAttachment(doc.toString(),
"Locks.htm", EmailAttachment.HTMLTEXT, "", EmailAttachment.ATTACHMENT)});
}

```

**Note:** Sending an e-mail puts an entry in the background process table showing the outcome of the e-mail.

## Internationalization

The framework provides a set of internationalization and localization utilities in the package `com.essbase.eas.i18n`. These classes provide a mechanism for locating resources associated with a window or dialog box, loading resource bundles based on the locale, localizing collections, arrays of components, or containers. There is also an i18n-friendly string collator class.

## Packaging the Plug-in

The only packaging requirement is that all classes and resources necessary for a client plug-in must be contained in the same jar file. You must include an entry in the jar file which defines the other jar files it depends on. For example, lets say the plug-in jar file `xyz.jar` depends on `abc.jar` and `cde.jar`, include the following entry in the manifest file for the plug-in jar file:

```
Class-Path: xyz.jar cde.jar
```



# 3

## Writing Server-side Command Listeners

### In This Chapter

Prerequisites.....	31
Command Listeners.....	32
Command Handling Methods.....	38
Packaging the Code .....	42
Loading the Code .....	43
Utility Classes .....	44

This chapter explains how to write a command listener for the Administration Services mid-tier web server. Installable command listeners are the mechanism for extending the functionality of the Administration Services Web server.

### Prerequisites

You should have the following skills and tools:

- You have some Java experience
- You have access to the *Administration Services Java API Reference*
- Since different developers use different build tools and environments, we do not discuss how to do anything for specific development environments. Rather, we describe the desired results, leaving it to the developer to know how to achieve these results with their specific development tools.

**Note:** For the purposes of this documentation, the terms “Administration Services web server”, “Administration Services servlet”, “Administration Services mid-tier”, “Administration Services framework”, and, simply, “the framework” can generally be taken to refer to the same object.

The **framework** is the Administration Services servlet and associated classes that receive commands, handle housekeeping duties, return results, and route commands to the registered listener.

## Command Listeners

A *command listener* is an instance of any class that implements the `CommandListener` interface; however, for practical purposes, all plug-in command listeners should extend one of these classes:

- `EssbaseCommandListener`
- `AppManCommandListener`
- `AbstractCommandListener`

The framework uses command listeners as the mechanism to properly route commands to be handled.

When the Administration Services servlet starts up, it builds a table of command listeners, the commands that each command listener can handle, and the method in the command listener for that command. As client applications send commands (http requests), the Administration Services servlet uses the command's operation parameter to determine the command listener and method to route the request to.

For example, a typical command might be to log in to the Administration Services servlet. When expressed as an http request, this command will look something like this:

```
http://LocalHost/EAS?op=login&name=user1&password=hello
```

The Administration Services servlet parses the following parameters:

- `op=login`
- `name=user1`
- `password=hello`

The framework uses the “op” parameter to route the command to the correct command listener. If the command listener has been registered correctly, the framework will also collect the “name=” and “password=” parameters and pass them as arguments to the method in the command listener.

## Class Hierarchy

The class hierarchy for the command listeners is:

```
com.essbase.eas.framework.server.application.AbstractCommandListener
com.essbase.eas.server.AppManCommandListener
com.essbase.eas.essbase.server.EssbaseCommandListener
```

All three of these classes are declared as abstract. You must extend from one of these three classes in order to have the framework find your command listener.

The `AbstractCommandListener` class provides the basic functionality that is needed for the framework. Most of the methods in this class are either final or protected; for most practical purposes, implementers of derived classes should not override the protected methods of this class. For a description of those methods that can be useful to implement in a derived class, see the section [“Which Methods to Override” on page 33](#).

The `AppManCommandListener` class adds some small functionality to the `AbstractCommandListener`, mostly dealing with EAS servlet session validation and exception handling during command routing.

The `EssbaseCommandListener` class adds some Essbase-specific functionality, primarily Oracle Essbase session validation.

## Which Class To Extend

Do not extend the `AbstractCommandListener` class, even though it is declared public. The `EssbaseCommandListener.handleEventPrep()` method checks some standard parameters for an Essbase Server name, application name, and database name and ensures a connection to that database if those parameters exist. If the implementer of the new command listener wishes to take advantage of the session handling performed by the `EssbaseCommandListener`, then they should extend this class; however, if this isn't necessary, the new command listener can extend the `AppManCommandListener` class.

## Which Methods to Override

`AbstractCommandListener.getCommands()` must be overridden. We explain more about this method in the section, [“Registering Commands” on page 34](#).

The `handleEventPrep()`, `handleEventPost()`, and `handleEventException()` methods may be overridden. These three methods, along with `AbstractCommandListener.handleEvent()`, form the core processing for any command received by the framework.

Once the framework determines which command listener to route a command to, it calls that command listener's `handleEvent()` method. Since the `AbstractCommandListener` declares this method as final, the framework always calls the method in `AbstractCommandListener`. This method then performs the following sequence of steps:

1. Calls `handleEventPrep()`; if this method returns true, then continues with step 2.
2. Gets the command listener's method that handles this specific command. If this method cannot be located, logs an error with the logging utility.
3. Converts the arguments from the http command into an array of Java objects.
4. Using Java introspection, invokes the method.
5. If no exceptions were thrown, invokes `handleEventPost()`.
6. If exceptions were thrown in steps 4 or 5, calls `handleEventException()`.

Any change to the processing of events before they arrive at a specific method in the command listener must be done by overriding the `handleEventPrep()` method. For instance, this is where the `EssbaseCommandListener` class checks Essbase sessions and the `AppManCommandListener` checks for a valid servlet session.

In most cases, the `handleEventPost()` method is empty and the `handleEventException()` method is empty.

## Registering Commands

After a command listener is instantiated by the framework, the framework calls the `getCommands()` method. This method returns an array of `CommandDescriptor` objects. The `CommandDescriptor` objects describe each command that the `CommandListener` is designed to handle. The `CommandDescriptor` object consists of three main parts:

- A string for the command
- The method in the command listener to call
- The list of arguments expected for this command.

The next few sections describe the classes used by the framework when registering commands.

**Note:** All of these classes are in the package `com.essbase.eas.framework.defs.command`.

### CommandString Class

A command listener handles commands like “GetDatabaseList”, “GetUsers”, “DeleteUsers”, and so on. The `CommandString` class was introduced to let each command listener programmer think of their commands in the simplest way. The `CommandString` class is declared as:

```
public abstract class CommandString
```

The only constructors are declared as:

```
private CommandString() { ... }  
protected CommandString(String original) { ... }
```

These two declarations combined mean that instances of this class can never be instantiated and derived classes must call the `CommandString(String original)` constructor with a valid `String` object as the parameter.

The most important action that instances of this class do is take the original `String` object and prepend the class name, including the package name, to the front of the `String`. This new value is then returned when the object’s `toString()` method is called.

### CommandArgument Class

The `CommandArgument` class describes individual arguments to commands. It contains the following fields:

- `String name` (available through the `getName()` method)  
This is the name of the http parameter corresponding to this argument.
- `boolean required` (available through the `isRequired()` method)  
Indicates whether this argument is required. The intent is that the framework can check this field when routing a command and return a pre-defined error status to the client if a required field is missing.
- `Class ClassType` (available through the `getClassType()` method)

This is used so the framework can convert the incoming text value to an appropriate object type.

- Object defaultValue (available through the getDefaultValue() method)

The framework will substitute this object for the argument if the argument is missing from the command.

- Boolean hidden (available through the isHidden() method)

The framework can log the retrieval and routing of commands and their parameters. Setting this field to true means the framework will not echo the value of this argument in the log file. This would be useful for passwords, and so on.

These fields are all declared as private and, since there are no setXXX() methods, cannot be changed after a CommandArgument object is constructed.

## CommandDescriptor Class

The CommandDescriptor class combines the CommandArgument and CommandString classes into a cohesive value so that the framework can construct its internal tables and route the commands as they are received.

The examples in the following sections show how all of this fits together.

## Examples

This section includes the following sample code:

- [Example](#)
- [Example](#)
- [Example](#)
- [Example](#)

### Example.java

---

```
// this is a simple class used as a parameter to show how the
// framework can separate out command arguments that are object
// types embedded in XML. For more information on how the
// framework uses XML to transport "generic" objects between the
// mid-tier and the client, please see the Java Docs references
// for the XMLTransferObject class.
public Example extends Object {
    private String name = "";
    private String[] text = new String[0];
    // no-argument constructor. Must be public for XML Transfer
    // to work.
    public Example() {
    }

    public String getName() {
        return name;
    }
}
```

```

}

public void setName(String value) {
    name = value;
}

public String[] getSampleText() {
    String[] result = new String[text.length];
    for (int i = 0; i < result.length; ++i)
        result[i] = text[i];
    return result;
}

public void setSampleText(String[] values) {
    if (values != null) {
        text = new String[values.length];
        for (int i = 0; i < values.length; ++i)
            text[i] = values[i];
    }
    else {
        text = new String[0];
    }
}
}

```

---

## ExampleCommandString.java

```

public ExampleCommandString extends CommandString {
    // declare some static String objects in a way that we know
    these
    // objects do not need to be translated to different locales.
    public static final String GET_EXAMPLES_TEXT = "GetExamples";
    public static final String ADD_EXAMPLE_TEXT = "AddExample";
    public static final String DELETE_EXAMPLE_TEXT =
"DeleteExample";

    // now we declare the actual commands
    public static final ExampleCommandString GET_EXAMPLES =
        new ExampleCommandString(GET_EXAMPLES_TEXT);
    public static final ExampleCommandString ADD_EXAMPLE =
        new ExampleCommandString(ADD_EXAMPLE_TEXT);
    public static final ExampleCommandString DELETE_EXAMPLE =
        new ExampleCommandString(DELETE_EXAMPLE_TEXT);

    // for organizational purposes, we also declare the parameters for each
    // of these commands in this file.
    public static final String PARAM_LOCATION = "location";
    public static final String PARAM_EXAMPLE = "example";
    public static final String PARAM_NAME = "examplename";

    // declare a CommandArgument object for each of these parameters
    private static final CommandArgument ARGUMENT_LOCATION =
        new CommandArgument(PARAM_LOCATION,
            true,
            String.class,
            null);
    private static final CommandArgument ARGUMENT_EXAMPLE =

```

```

        new CommandArgument(PARAM_EXAMPLE,
            true,
            Example.class,
            null);
private static final CommandArgument ARGUMENT_NAME =
    new CommandArgument(PARAM_NAME,
        true,
        String.class,
        null);

// declare an array of arguments for each command.
public static final CommandArgument[] GET_EXAMPLES_ARGS =
    new CommandArgument[] { ARGUMENT_LOCATION };
public static final CommandArgument[] ADD_EXAMPLE_ARGS =
    new CommandArgument[] { ARGUMENT_LOCATION,
        ARGUMENT_EXAMPLE };
public static final CommandArgument[] DELETE_EXAMPLE_ARGS =
    new CommandArgument[] { ARGUMENT_LOCATION,
        ARGUMENT_NAME };
}

```

---

This class declares command strings and describes the arguments for three commands that will be supported by the `ExampleCommandListener` class. If the `toString()` method of each `ExampleCommandString` object declared in this source code file were called, the results would be:

```

ExampleCommandString.GetExamples
ExampleCommandString.AddExample
ExampleCommandString.DeleteExample

```

Every `CommandDescriptor` object contains a reference to an object derived from `CommandString`; it is through this mechanism that the framework guarantees every command name is unique.

## ExampleDescriptor.java

---

```

public class ExampleDescriptor extends CommandDescriptor {
    private static final String GET_EXAMPLES_METHOD = "getExamples";
    private static final String ADD_EXAMPLE_METHOD = "addExample";
    private static final String DELETE_EXAMPLE_METHOD = "deleteExample";

    public static final CommandDescriptor GET_EXAMPLES =
        new CommandDescriptor(ExampleCommands.GET_EXAMPLES,
            GET_EXAMPLES_METHOD,
            ExampleCommands.GET_EXAMPLES_ARGS);
    public static final CommandDescriptor ADD_EXAMPLE =
        new CommandDescriptor(ExampleCommands.ADD_EXAMPLE,
            ADD_EXAMPLE_METHOD,
            ExampleCommands.ADD_EXAMPLE_ARGS);
    public static final CommandDescriptor DELETE_EXAMPLE =
        new CommandDescriptor(ExampleCommands.DELETE_EXAMPLE,
            DELETE_EXAMPLE_METHOD,
            ExampleCommands.DELETE_EXAMPLE_ARGS);
}

```

---

## ExampleCommandListener.java

---

```
public class ExampleCommandListener extends AppManCommandListener {
    // the method called when the GetExamples command is received.
    public boolean getExamples(CommandEvent theEvent,
        ServiceContext theContext,
        String theLocation) {
        // the details will be filled in later
        return true;
    }

    // the method called when the AddExample command is received.
    public boolean addExample(CommandEvent theEvent,
        ServiceContext theContext,
        String theLocation,
        Example theExample) {
        // the details will be filled in later
        return true;
    }

    // the method called when the DeleteExample command is
    // received.
    public boolean deleteExample(CommandEvent theEvent,
        ServiceContext theContext,
        String theLocation,
        String theName) {
        // the details will be filled in later.
        return true;
    }

    // the framework calls this method to get the descriptors for
    // the commands supported by this command listener.
    public CommandDescriptor[] getCommands() {
        return new CommandDescriptor[] {
            ExampleDescriptor.GET_EXAMPLES,
            ExampleDescriptor.ADD_EXAMPLE,
            ExampleDescriptor.DELETE_EXAMPLE };
    }
}
```

---

The preceding example shows the skeleton of a command listener:

1. Extend the correct class
2. Add the command handling methods
3. Override the `getCommands()` method to return the descriptors for those commands.

The difficulty is in the details of the command handling methods, which is covered in the next section.

## Command Handling Methods

This section includes the following topics:

- [“Method Signatures” on page 39](#)

- “Grabbing Command Arguments” on page 40
- “Sending Results Back to the Client” on page 40
- “Storing Temporary Data Using the Framework” on page 41

## Method Signatures

If you were looking carefully at the example code in the preceding section, you might be saying something along the lines of, “Wait a minute, in GET\_EXAMPLES\_ARGS, I defined one argument, the location argument. What are these other two arguments, theEvent and theContext? Where did they come from and what do I do with them?” The answer partly lies in the older version of the Administration Services framework. The first version of the framework did not do all the type checking and parameter parsing that the new level does, so all command handling methods had the following signature:

```
public boolean handlerMethod(CommandEvent theEvent) { }
```

It was up to each method to then extract and handle the arguments along the lines of:

```
String theLocation = theEvent.getRawParameter("Location");
if (theLocation == null) {
    // oops - this shouldn't happen!
    return false;
}
```

Or, if the parameter was supposed to be a numeric value:

```
int theNumber = 0;
String theValue = theEvent.getRawParameter("Value");
if (theValue == null) {
    // oops - this shouldn't happen!
    return false;
}
try {
    theNumber = Integer.parseInt(theValue)
}
catch (Exception ex) {
    return false;
}
```

In most cases, theEvent object was used mostly to get the parameters for the command. When the framework was upgraded, theEvent object was retained as the first argument to the command handler methods, even though it is rarely used.

The second argument, theContext, is actually a field in theEvent object; if you want to return results to the client, you must do so through the ServiceContext reference. Since every command handling method at some time would call theEvent.getServiceContext(), we decided to add it as a second parameter to every command handling method.

As a result of these decisions, every command handling method has the following signature:

```
public boolean handlerMethod(CommandEvent theEvent,
    ServiceContext theContext,
    Class0 value0,
```

```
...  
ClassN, ValueN);
```

Where the ClassX parameters are described by the CommandDescriptor for the method.

In addition, even though the method is declared boolean, the framework never looks at the return value from a command handler method. Return values are handled within each method by a mechanism explained later in this document.

## Grabbing Command Arguments

In most cases, the command arguments will have been extracted and parsed by the framework; however, special circumstances can arise whereby extra arguments are sent with each command that, for whatever reason, the programmer doesn't want to include in the CommandDescriptor object.

An example is the EssbaseCommandListener; the EssbaseCommandListener.handleEventPrep() method calls a validateSession() method that looks for the standard parameters “servername”, “appname”, “dbname”, then attempts to validate an EssbaseSession against those parameters. If this fails, then the handleEventPrep() method returns a standard error status to the client. In most cases, any EssbaseCommandListener will need these arguments when handling commands. However, there are cases (such as in the outline editor) when those arguments aren't used. If, during implementation of a command listener method, a similar situation arises, the parameters can be retrieved by the following call:

```
String aValue = theEvent.getRawParameter("SomeParameterName");
```

This should be a rare necessity and should raise caution alarms if an implementer finds themselves needing to do this.

## Sending Results Back to the Client

There are two types of results to return to a client:

1. Status of the command
2. Data that the client needs to display

The CommandStatus class is used to return the success/failure of the command to the client. The CommandStatus class only understands two types of status: SUCCESS and FAILURE. The original intent of this class was to indicate whether a command was routed successfully by the framework. However, this wasn't made explicit and, as a result, many existing command handling methods use this SUCCESS/FAILURE to indicate the status of their specific processing.

It would be a good practice to always extend this class to enable returning more specific error codes than just SUCCESS/FAILURE.

So, let's return to our example and fill in one of the command handling methods to return data and a SUCCESSFUL status to the client.

```
public boolean getExamples(CommandEvent theEvent,  
    ServiceContext theContext,
```

```

        String theLocation) {
//object used to transmit results back to the client
XMLTransferObject xto=new XMLTransferObject();
Example [] theResults=someMethod(theLocation);
if (theResults == null) {
    //this is simplistic, but it shows what we need
    xto.setCommandStatus(CommandStatus.SIMPLE_FAILURE);
}
else {
    if (theResults.length != 0)
        xto.addAll(theResults);
    xto.setCommandStatus(CommandStatus.SIMPLE_SUCCESS);
}
this.storeService.set(theContext,
    DefaultScopeType.REQUEST_SCOPE,
    AppManServlet.RESULT,
    xto.exportXml());
return true;
}

```

The XMLTransferObject is used to transmit the data and the command status back to the client; we use the defined CommandStatus.SIMPLE\_FAILURE or CommandStatus.SIMPLE\_SUCCESS objects to return the correct status. If results were available, they were then added to the XMLTransferObject using the addAll() method. The results were then placed in the command listener's store service using the REQUEST\_SCOPE and using the key AppManServlet.RESULT. After this method returns to the framework, the framework will take any data stored using the combination DefaultScopeType.REQUEST\_SCOPE and AppManServlet.RESULT and send that data back to the client as the results of the command.

## Storing Temporary Data Using the Framework

In the preceding section, we gave an example of how to place data in the framework's storage so that the data would be returned to the client as the results of a command. The storeService field in each command manager can store data for additional purposes. There are six defined DefaultScopeTypes:

1. CONFIG\_SCOPE

This is used by the framework as it is initializing. It should never be used by command handler methods.

2. BUILDER\_SCOPE

This is used by the framework as it is initializing. It should never be used by command handler methods.

3. APP\_SCOPE

Using this scope type will cause the data to be stored for the life of the servlet. This should be very, very rarely used by command listeners.

4. SESSION\_SCOPE

Using this scope type will cause the data to be stored until the current client/server session is no longer valid. At that point, the framework will remove all data stored in this scope. Store information in this scope that needs to be recovered when processing subsequent commands.

#### 5. USER\_SCOPE

Using this scope makes the data available to any client connected using the same EAS user id. When all sessions associated with this user are no longer valid, the framework will remove data stored in this scope. In the current implementation, this is never used and it probably will never be used very often.

#### 6. REQUEST\_SCOPE

Using this scope makes the data available until the framework has bundled the results of the command and returned them to the client. The framework then removes all data stored in this scope associated with the request that just finished.

Storing data is done through a command listener's store service, as in the preceding example. The StoreService interface has several get(), set(), and remove() methods. However, there is only one of each of these methods that a command listener (or other plug-in code) should call; the other methods were put in place for use by some of the framework code itself. The three method signatures are:

```
public Object get(ServiceContext context, ScopeType type, Object key);
public Object set(ServiceContext context, ScopeType type, Object key, Object value);
public Object remove(ServiceContext context, ScopeType type, Object key);
```

For more information about these methods, see the *Essbase Administration Services Java API Reference*.

## Packaging the Code

When packaging the code into jar files for a plug-in, follow these guidelines:

- Separate the code into three distinct pieces:
  - Code that is only used on the client
  - Code that is only used on the server
  - Code that is used in both places
- Set up the build tools to compile and package these pieces separately to prevent crossover compilation. For example, the framework is packaged into the following jar files:

```
framework_client.jar
framework_common.jar
framework_server.jar
```

- Package the command listener classes in the server jar
- Package the command descriptor classes in the server jar. This is because they contain references to the method names in the command listeners and this should not be publicly available on the client.

- Package the `CommandString` derived classes in the common jar file. While the framework does not currently take advantage of this on the client, it will be upgraded to do the packaging of parameters and commands for client applications.
- Place any classes extending `CommandStatus` in the common jar file.
- Place any specialized classes (such as `Example.java`) in the common jar file.

The server jar file must contain a manifest file. Each command listener must have an entry in this manifest file that looks like the following:

```
Name: ExampleCommandListener.class
```

```
EAS-Framework-CommandListener: True
```

If, as is likely, the command listener has a package name that must be prepended to the name in the example above, like this:

```
Name: com/essbase/eas/examples/server/ExampleCommandListener.class
```

```
EAS-Framework-CommandListener: True
```

**Note:** Even though this is a class name, use slashes (“/”) instead of dots (“.”) to separate the package names.

## Loading the Code

To enable the framework to recognize command listeners and route commands to the correct place, the jar file containing the command listeners and any other jar files on which this code depends must be bundled inside Administration Server’s `eas.war` file. The `eas.war` file is contained in the `eas.ear` file, and must be unpacked so that the command listeners can be added.

► To bundle the command listeners:

- 1 **Unzip the `eas.ear` file.**
- 2 **Unzip the `eas.war` file.**
- 3 **In the directory structure thus created, add the command listeners to the `WEB-INF/lib` directory.**
- 4 **Rezip the `eas.war` file.**
- 5 **Rezip the `eas.ear` file.**

**Note:** The `eas.ear` file is installed by default to `EPM_ORACLE_HOME\products\Essbase\eam\server\AppServer\InstallableApps\Common`.

After putting the jar files in this location, you must stop and restart Administration Server. To determine if the new command listeners have been installed, set the Administration Services logging level between `INFO` and `ALL`.

## Utility Classes

There are many utility classes provided by the Oracle Essbase Administration Services framework. In particular, there are utility classes in some of the following packages:

```
com.essbase.eas.framework.defs  
com.essbase.eas.framework.server  
com.essbase.eas.utils  
com.essbase.eas.ui  
com.essbaes.eas.i18n  
com.essbase.eas.net
```

The *Administration Services Java API Reference* makes it easy to navigate through these classes and learn what is available.

---

# Glossary

---

! See bang character.

**#MISSING** See missing data.

**access permissions** A set of operations that a user can perform on a resource.

**accessor** Input and output data specifications for data-mining algorithms.

**accounts dimension** A dimension type that makes accounting intelligence available. Only one dimension can be defined as Accounts.

**Advanced Relational Access** The integration of a relational database with an Essbase multidimensional database so that all data remains in the relational database and is mapped to summary-level data in the Essbase database.

**agent** An Essbase server process that starts and stops applications and databases, manages connections from users, and handles user-access security. The agent is referred to as ESSBASE.EXE.

**aggregate cell** A cell comprising several cells. For example, a data cell that uses Children(Year) expands to four cells containing Quarter 1, Quarter 2, Quarter 3, and Quarter 4 data.

**aggregate storage database** The database storage model designed to support large-scale, sparsely distributed data which is categorized into many, potentially large dimensions. Upper level members and formulas are dynamically calculated, and selected data values are aggregated and stored, typically with improvements in overall aggregation time.

**aggregate view** A collection of aggregate cells based on the levels of the members within each dimension. To reduce calculation time, values are pre-aggregated and stored as aggregate views. Retrievals start from aggregate view totals and add up from there.

**aggregation** The process of rolling up and storing values in an aggregate storage database; the stored result of the aggregation process.

**aggregation script** In aggregate storage databases only, a file that defines a selection of aggregate views to be built into an aggregation.

**alternate hierarchy** A hierarchy of shared members. An alternate hierarchy is based upon an existing hierarchy in a database outline, but has alternate levels in the dimension. An alternate hierarchy allows the same data to be seen from different points of view.

**ancestor** A branch member that has members below it. For example, the members Qtr2 and 2006 are ancestors of the member April.

**application** 1) A software program designed to run a specific task or group of tasks such as a spreadsheet program or database management system; 2) A related set of dimensions and dimension members that are used to meet a specific set of analytical requirements, reporting requirements, or both.

**area** A predefined set of members and values that makes up a partition.

**arithmetic data load** A data load that performs operations on values in the database, such as adding 10 to each value.

**artifact** An individual application or repository item; for example, scripts, forms, rules files, Interactive Reporting documents, and financial reports. Also known as an object.

**attribute** A characteristic of a dimension member. For example, Employee dimension members may have attributes of Name, Age, or Address. Product dimension members can have several attributes, such as a size and flavor.

**attribute association** A relationship in a database outline whereby a member in an attribute dimension describes a characteristic of a member of its base dimension. For example, if product 100-10 has a grape flavor, the product 100-10 has the Flavor attribute association of grape. Thus, the 100-10 member of the Product dimension is associated with the Grape member of the Flavor attribute dimension.

**Attribute Calculations dimension** A system-defined dimension that performs these calculation operations on groups of members: Sum, Count, Avg, Min, and Max. This dimension is calculated dynamically and is not visible in the database outline. For example, using the Avg member, you can calculate the average sales value for Red products in New York in January.

**attribute dimension** A type of dimension that enables analysis based on the attributes or qualities of dimension members.

**attribute reporting** A reporting process based on the attributes of the base dimension members. See also base dimension.

**attribute type** A text, numeric, Boolean, date, or linked-attribute type that enables different functions for grouping, selecting, or calculating data. For example, because the Ounces attribute dimension has the type numeric, the number of ounces specified as the attribute of each product can be used to calculate the profit per ounce for that product.

**authentication** Verification of identity as a security measure. Authentication is typically based on a user name and password. Passwords and digital signatures are forms of authentication.

**axis** 1) A straight line that passes through a graphic used for measurement and categorization; 2) A report aspect used to arrange and relate multidimensional data, such as filters, pages, rows, and columns. For example, for a data query in Simple Basic, an axis can define columns for values for Qtr1, Qtr2, Qtr3, and Qtr4. Row data would be retrieved with totals in the following hierarchy: Market, Product.

**bang character (!)** A character that terminates a series of report commands and requests information from the database. A report script must be terminated with a bang character; several bang characters can be used within a report script.

**base currency** The currency in which daily business transactions are performed.

**base dimension** A standard dimension that is associated with one or more attribute dimensions. For example, assuming products have flavors, the Product dimension is the base dimension for the Flavors attribute dimension.

**batch calculation** Any calculation on a database that is done in batch; for example, a calculation script or a full database calculation. Dynamic calculations are not considered to be batch calculations.

**batch file** An operating system file that can call multiple ESSCMD scripts and run multiple sessions of ESSCMD. On Windows-based systems, batch files have BAT file extensions. On UNIX, batch files are written as a shell script.

**batch processing mode** A method of using ESSCMD to write a batch or script file that can be used to automate routine server maintenance and diagnostic tasks. ESSCMD script files can execute multiple commands and can be run from the operating system command line or from within operating system batch files. Batch files can be used to call multiple ESSCMD scripts or run multiple instances of ESSCMD.

**block** The primary storage unit which is a multidimensional array representing the cells of all dense dimensions.

**block storage database** The Essbase database storage model categorizing and storing data based on the sparsity of data values defined in sparse dimensions. Data values are stored in blocks, which exist only for sparse dimension members for which there are values.

**build method** A method used to modify database outlines. Choice of a build method is based on the format of data in data source files.

**cache** A buffer in memory that holds data temporarily.

**calculated member in MaxL DML** A member designed for analytical purposes and defined in the optional WITH section of a MaxL DML query.

**cascade** The process of creating multiple reports for a subset of member values.

**CDF** See custom-defined function.

**CDM** See custom-defined macro.

**cell** 1) The data value at the intersection of dimensions in a multidimensional database; the intersection of a row and a column in a worksheet; 2) A logical group of nodes belonging to one administrative domain.

**cell note** A text annotation for a cell in an Essbase database. Cell notes are a type of LRO.

**child** A member with a parent above it in the database outline.

**clean block** A data block in which the database is fully calculated, if a calculation script calculates all dimensions at once, or if the SET CLEARUPDATESTATUS command is used in a calculation script.

**code page** A mapping of bit combinations to a set of text characters. Different code pages support different sets of characters. Each computer contains a code page setting for the character set requirements of the language of the computer user. In the context of this document, code pages map characters to bit combinations for non-Unicode encodings. See also encoding.

**committed access** An Essbase Kernel Isolation Level setting that affects how Essbase handles transactions. Under committed access, concurrent transactions hold long-term write locks and yield predictable results.

**consolidation** The process of aggregating data from dependent entities to parent entities. For example, if the dimension Year consists of the members Qtr1, Qtr2, Qtr3, and Qtr4, its consolidation is Year.

**crosstab reporting** Reporting that categorizes and summarizes data in table format. The table cells contain summaries of the data that fit within the intersecting categories. For example, a crosstab report of product sales information could show size attributes, such as Small and Large, as column headings and color attributes, such as Blue and Yellow, as row headings. The cell in the table where Large and Blue intersect could contain the total sales of all Blue products that are sized Large.

**cube** A block of data that contains three or more dimensions. An Essbase database is a cube.

**cube deployment** In Essbase Studio, the process of setting load options for a model to build an outline and load data into an Essbase application and database.

**cube schema** In Essbase Studio, the metadata elements, such as measures and hierarchies, representing the logical model of a cube.

**currency conversion** A process that converts currency values in a database from one currency into another. For example, to convert one U. S. dollar into the European euro, the exchange rate (for example, 0.923702) is multiplied by the dollar (1\* 0.923702). After conversion, the European euro amount is .92.

**currency partition** A dimension type that separates local currency members from a base currency, as defined in an application. Identifies currency types, such as Actual, Budget, and Forecast.

**custom-defined function (CDF)** Essbase calculation functions developed in Java and added to the standard Essbase calculation scripting language using MaxL. See also custom-defined macro.

**custom-defined macro (CDM)** Essbase macros written with Essbase calculator functions and special macro functions. Custom-defined macros use an internal Essbase macro language that enables the combination of calculation functions and they operate on multiple input parameters. See also custom-defined function.

**cycle through** Perform multiple passes through a database while calculating it.

**data cache** A buffer in memory that holds uncompressed data blocks.

**data cell** See cell.

**data file cache** A buffer in memory that holds compressed data (PAG) files.

**data load rules** A set of criteria that determines how to load data from a text-based file, a spreadsheet, or a relational data set into a database.

**data mining** The process of searching through an Essbase database for hidden relationships and patterns in a large amount of data.

**data value** See cell.

**date measure** In Essbase, a member tagged as Date in the dimension where measures are represented. The cell values are displayed as formatted dates. Dates as measures can be useful for analysis types that are difficult to represent using the Time dimension. For example, an application may need to track acquisition dates for a series of capital assets, but the acquisition dates span too large a period to allow for feasible Time dimension modeling. See also typed measure.

**dense dimension** In block storage databases, a dimension likely to contain data for every combination of dimension members. For example, time dimensions are often dense because they can contain all combinations of all members. Contrast with sparse dimension.

**derived text measure** In Essbase Studio, a text measure whose values are governed by a predefined rule expressed as a range. For example, a derived text measure, called "Sales Performance Index," based on a measure Sales, could consist of the values "High," "Medium," and "Low." This derived text measure is defined to display "High," "Medium," and "Low" depending on the range in which the corresponding sales values fall. See also text measure.

**descendant** Any member below a parent in the database outline. In a dimension that includes years, quarters, and months, the members Qtr2 and April are descendants of the member Year.

**dimension** A data category used to organize business data for the retrieval and preservation of values. Dimensions usually contain hierarchies of related members grouped within them. For example, a Year dimension often includes members for each time period, such as quarters and months.

**dimension build** The process of adding dimensions and members to an Essbase outline.

**dimension build rules** Specifications, similar to data load rules, that Essbase uses to modify an outline. The modification is based on data in an external data source file.

**dimension type** A dimension property that enables the use of predefined functionality. Dimensions tagged as time have a predefined calendar functionality.

**dimensionality** In MaxL DML, the represented dimensions (and the order in which they are represented) in a set. For example, the following set consists of two tuples of the same dimensionality, because they both reflect the dimensions (Region, Year): { (West, Feb), (East, Mar) }

**dirty block** A data block containing cells that have been changed since the last calculation. Upper-level blocks are marked as dirty if their child blocks are dirty (that is, if they have been updated).

**domain** In data mining, a variable representing a range of navigation within data.

**drill-down** Navigation through the query result set using the dimensional hierarchy. Drilling down moves the user perspective from aggregated data to detail. For example, drilling down can reveal hierarchical relationships between years and quarters or quarters and months.

**duplicate alias name** A name that occurs more than once in an alias table and can be associated with more than one member in a database outline. Duplicate alias names can be used with duplicate member outlines only.

**duplicate member name** Multiple occurrences of a member name in a database, with each occurrence representing a different member. For example, a database has two members named New York. One member represents New York state and the other member represents New York city.

**duplicate member outline** A database outline containing duplicate member names.

**Dynamic Calc and Store members** Members in a block storage outline that Essbase calculates only upon the first retrieval of the value. Essbase then stores the calculated value in the database. Subsequent retrievals do not require calculating.

**Dynamic Calc members** Members in a block storage outline that Essbase calculates only at retrieval time. Essbase discards calculated values after completing the retrieval request.

**dynamic calculation** In Essbase, a calculation that occurs only when you retrieve data on a member that is tagged as Dynamic Calc or Dynamic Calc and Store. The member's values are calculated at retrieval time instead of being precalculated during batch calculation.

**dynamic hierarchy** In aggregate storage database outlines only, a hierarchy in which members are calculated at retrieval time.

**dynamic reference** A pointer in the rules file to header records in a data source.

**Dynamic Time Series** A process that performs period-to-date reporting in block storage databases.

**encoding** A method for mapping bit combinations to characters for creating, storing, and displaying text. Each encoding has a name; for example, UTF-8. Within an encoding, each character maps to a specific bit combination; for example, in UTF-8, uppercase A maps to HEX41. See also code page, locale.

**Enterprise View** An Administration Services feature that enables management of the Essbase environment from a graphical tree view. From Enterprise View, you can operate directly on Essbase artifacts.

**essbase.cfg** An optional configuration file for Essbase. Administrators may edit this file to customize Essbase Server functionality. Some configuration settings may also be used with Essbase clients to override Essbase Server settings.

**EssCell** A function entered into a cell in Essbase Spreadsheet Add-in to retrieve a value representing an intersection of specific Essbase database members.

**ESSCMD** A command-line interface for performing Essbase operations interactively or through batch script files.

**ESSLANG** The Essbase environment variable that defines the encoding used to interpret text characters. See also encoding.

**ESSMSH** See MaxL Shell.

**external authentication** Logging on to Oracle EPM System products with user information stored outside the application. The user account is maintained by the EPM System, but password administration and user authentication are performed by an external service, using a corporate directory such as Oracle Internet Directory (OID) or Microsoft Active Directory (MSAD).

**extraction command** An Essbase reporting command that handles the selection, orientation, grouping, and ordering of raw data extracted from a database; begins with the less-than (<) character.

**file delimiter** A character, such as a comma or tab, that separates fields in a data source.

**filter** A constraint on data sets that restricts values to specific criteria; for example, to exclude certain tables, metadata, or values, or to control access.

**format string** 1) In Essbase, a method for transforming the way cell values are displayed; 2) In Data Relationship Management, a parameter of a Format or Formatted Date derived property that indicates the format in which a property value should be returned.

**free-form reporting** Creating reports by entering dimension members or report script commands in worksheets.

**generation** A layer in a hierarchical tree structure that defines member relationships in a database. Generations are ordered incrementally from the top member of the dimension (generation 1) down to the child members. Use the unique generation name to identify a layer in the hierarchical tree structure.

**global report command** A command in a running report script that is effective until it is replaced by another global command or the file ends.

**GUI** Graphical user interface

**Hybrid Analysis** An analysis mapping low-level data stored in a relational database to summary-level data stored in Essbase, combining the mass scalability of relational systems with multidimensional data.

**index** 1) A method where Essbase uses sparse-data combinations to retrieve data in block storage databases. 2) The index file.

**index cache** A buffer containing index pages.

**index entry** A pointer to an intersection of sparse dimensions. Index entries point to data blocks on disk and use offsets to locate cells.

**index file** An Essbase file storing block storage data retrieval information, residing on disk, and containing index pages.

**index page** A subdivision in an index file. An index page contains pointers to data blocks.

**input data** Data loaded from a source rather than calculated.

**intelligent calculation** A calculation method tracking updated data blocks since the last calculation.

**interdimensional irrelevance** A situation in which a dimension does not intersect with other dimensions. Because the data in the dimension cannot be accessed from the nonintersecting dimensions, the nonintersecting dimensions are not relevant to that dimension.

**introspection** A deep inspection of a data source to discover hierarchies based on the inherent relationships in the database. Contrast with scraping.

**isolation level** An Essbase Kernel setting that determines the lock and commit behavior of database operations. Choices are: committed access and uncommitted access.

**layer** 1) The horizontal location of members in a hierarchical structure, specified by generation (top down) or level (bottom up); 2) Position of objects relative to other objects. For example, in the Sample Basic database, Qtr1 and Qtr4 are in the same layer, so they are also in the same generation, but in a database with a ragged hierarchy, Qtr1 and Qtr4 might not be in same layer, though they are in the same generation.

**level** A layer in a hierarchical tree structure that defines database member relationships. Levels are ordered from the bottom dimension member (level 0) up to the parent members.

**level 0 block** A data block for combinations of sparse, level 0 members.

**level 0 member** A member that has no children.

**lineage** The relationship between different metadata elements showing how one metadata element is derived from one or more other metadata elements, ultimately tracing the metadata element to its physical source. In Essbase Studio, a lineage viewer displays the relationships graphically. See also traceability.

**link** 1) A reference to a repository object. Links can reference folders, files, shortcuts, and other links; 2) In a taskflow, the point where the activity in one stage ends and another begins.

**linked partition** A shared partition that enables you to use a data cell to link two databases. When a user clicks a linked cell in a worksheet, Essbase opens a new sheet displaying the dimensions in the linked database. The user can then drill down those dimensions.

**linked reporting object (LRO)** A cell-based link to an external file such as cell notes, URLs, or files with text, audio, video, or pictures. (Only cell notes are supported for Essbase LROs in Financial Reporting.) Contrast with local report object.

**locale** A computer setting that specifies a location's language, currency and date formatting, data sort order, and the character set encoding used on the computer. Essbase uses only the encoding portion. See also encoding, ESSLANG.

**locale header record** A text record at the beginning of some non-Unicode-encoded text files, such as scripts, that identifies the encoding locale.

**location alias** A descriptor that identifies a data source. The location alias specifies a server, application, database, user name, and password. Location aliases are set by DBAs at the database level using Administration Services Console, ESSCMD, or the API.

**Log Analyzer** An Administration Services feature that enables filtering, searching, and analysis of Essbase logs.

**LRO** See linked reporting object.

**mathematical operator** A symbol that defines how data is calculated in formulas and outlines. Can be any of the standard mathematical or Boolean operators; for example, +, -, \*, /, and %.

**MaxL** The multidimensional database access language for Essbase, consisting of a data definition language (MaxL DDL) and a data manipulation language (MaxL DML). See also MaxL DDL, MaxL DML, and MaxL Shell

**MaxL DDL** The data definition language used by Essbase for batch or interactive system-administration tasks.

**MaxL DML** The data manipulation language used in Essbase for data query and extraction.

**MaxL Perl Module** A Perl module (essbase.pm) that is part of Essbase MaxL DDL. This module can be added to the Perl package to provide access to Essbase databases from Perl programs.

**MaxL Script Editor** A script-development environment in Administration Services Console. MaxL Script Editor is an alternative to using a text editor and the MaxL Shell for administering Essbase with MaxL scripts.

**MaxL Shell** An interface for passing MaxL statements to Essbase Server. The MaxL Shell executable file is located in the Essbase bin directory (UNIX: essmsh; Windows: essmsh.exe).

**member** A discrete component within a dimension. A member identifies and differentiates the organization of similar units. For example, a time dimension might include members Jan, Feb, and Qtr1.

**member load** In Essbase Integration Services, the process of adding dimensions and members (without data) to Essbase outlines.

**member selection report command** A type of Report Writer command that selects member ranges based on outline relationships, such as sibling, generation, and level.

**member-specific report command** A type of Report Writer formatting command that is executed as it is encountered in a report script. The command affects only its associated member and executes the format command before processing the member.

**metadata elements** Metadata derived from data sources and other metadata that is stored and cataloged for Essbase Studio use.

**metadata sampling** The process of retrieving a sample of members in a dimension in a drill-down operation.

**metadata security** Security set at the member level to restrict users from accessing certain outline members.

**metaoutline** In Essbase Integration Services, a template containing the structure and rules for creating an Essbase outline from an OLAP model.

**mining attribute** In data mining, a class of values used as a factor in analysis of a set of data.

**minischema** A graphical representation of a subset of tables from a data source that represents a data modeling context.

**missing data (#MISSING)** A marker indicating that data in the labeled location does not exist, contains no value, or was never entered or loaded. For example, missing data exists when an account contains data for a previous or future period but not for the current period.

**model** 1) In data mining, a collection of an algorithm's findings about examined data. A model can be applied against a wider data set to generate useful information about that data; 2) A file or content string containing an application-specific representation of data. Models are the basic data managed by Shared Services, of two major types: dimensional and nondimensional application objects; 3) In Business Modeling, a network of boxes connected to represent and calculate the operational and financial flow through the area being examined.

**multidimensional database** A method of organizing, storing, and referencing data through three or more dimensions. An individual value is the intersection point for a set of dimensions. Contrast with relational database.

**named set** In MaxL DML, a set with its logic defined in the optional WITH section of a MaxL DML query. The named set can be referenced multiple times in the query.

**nested column headings** A report column heading format that displays data from multiple dimensions. For example, a column heading that contains Year and Scenario members is a nested column. The nested column heading shows Q1 (from the Year dimension) in the top line of the heading, qualified by Actual and Budget (from the Scenario dimension) in the bottom line of the heading.

**non-dimensional model** A Shared Services model type that includes application objects such as security files, member lists, calculation scripts, and Web forms.

**non-unique member name** See duplicate member name.

**numeric attribute range** A feature used to associate a base dimension member that has a discrete numeric value with an attribute that represents a value range. For example, to classify customers by age, an Age Group attribute dimension can contain members for the following age ranges: 0-20, 21-40, 41-60, and 61-80. Each Customer dimension member can be associated with an Age Group range. Data can be retrieved based on the age ranges rather than on individual age values.

**OLAP Metadata Catalog** In Essbase Integration Services, a relational database containing metadata describing the nature, source, location, and type of data that is pulled from the relational data source.

**OLAP model** In Essbase Integration Services, a logical model (star schema) that is created from tables and columns in a relational database. The OLAP model is then used to generate the structure of a multidimensional database. See also online analytical processing (OLAP).

**online analytical processing (OLAP)** A multidimensional, multiuser, client-server computing environment for users who analyze consolidated enterprise data in real time. OLAP systems feature drill-down, data pivoting, complex calculations, trend analysis, and modeling.

**outline** The database structure of a multidimensional database, including all dimensions, members, tags, types, consolidations, and mathematical relationships. Data is stored in the database according to the structure defined in the outline.

**outline synchronization** For partitioned databases, the process of propagating outline changes from one database to another database.

**page file** An Essbase data file.

**page heading** A report heading type that lists members represented on the current page of the report. All data values on the page have the members in the page heading as a common attribute.

**parallel calculation** A calculation option. Essbase divides a calculation into tasks and calculates some tasks simultaneously.

**parallel data load** In Essbase, the concurrent execution of data load stages by multiple process threads.

**parallel export** The ability to export Essbase data to multiple files. This may be faster than exporting to a single file, and it may resolve problems caused by a single data file becoming too large for the operating system to handle.

**partition area** A subcube within a database. A partition is composed of one or more areas of cells from a portion of the database. For replicated and transparent partitions, the number of cells within an area must be the same for the data source and target to ensure that the two partitions have the same shape. If the data source area contains 18 cells, the data target area must also contain 18 cells to accommodate the number of values.

**partitioning** The process of defining areas of data that are shared or linked between data models. Partitioning can affect the performance and scalability of Essbase applications.

**pattern matching** The ability to match a value with any or all characters of an item entered as a criterion. Missing characters may be represented by wild-card values such as a question mark (?) or an asterisk (\*). For example, "Find all instances of apple" returns apple, but "Find all instances of apple\*" returns apple, applesauce, applecranberry, and so on.

**periodicity** Any shared pattern among time-related dimension members that makes them meaningful for time-based analysis. For example, Jan and Apr share the periodicity of being opening months of a quarter in the Gregorian calendar.

**permission** A level of access granted to users and groups for managing data or other users and groups.

**persistence** The continuance or longevity of effect for any Essbase operation or setting. For example, an Essbase administrator may limit the persistence of user name and password validity.

**pivot** Alter the perspective of retrieved data. When Essbase first retrieves a dimension, it expands data into rows. You can then pivot or rearrange the data to obtain a different viewpoint.

**precalculation** Calculating the database before user retrieval.

**preserve formulas** User-created formulas kept within a worksheet while retrieving data.

**provisioning** The process of granting users and groups specific access permissions to resources.

**qualified name** A member name in a qualified format that differentiates duplicate member names in a duplicate member outline. For example, [Market].[East].[State].[New York] or [Market].[East].[City].[New York].

**query governor** An Essbase Integration Server parameter or Essbase Server configuration setting that controls the duration and size of queries made to data sources.

**ragged hierarchy** An asymmetrical hierarchy that occurs when a member has branches that contain different numbers of levels. For example, assume a Country dimension containing hierarchies of different geographical entities down to cities at level 0. The United States hierarchy contains three levels: country, state, and city. The Greece hierarchy contains two levels: country and city.

**record** In a database, a group of fields making up one complete entry. For example, a customer record may contain fields for name, address, telephone number, and sales data.

**redundant data** Duplicate data blocks that Essbase retains during transactions until Essbase commits updated blocks.

**replicated partition** A portion of a database, defined through Partition Manager, used to propagate an update to data mastered at one site to a copy of data stored at another site. Users can access the data as though it were part of their local database.

**Report Extractor** An Essbase component that retrieves report data from the Essbase database when report scripts are run.

**report script** A text file containing Essbase Report Writer commands that generate one or more production reports.

**Report Viewer** An Essbase component that displays complete reports after report scripts are run.

**restore** An operation to reload data and structural information after a database has been damaged or destroyed, typically performed after shutting down and restarting the database.

**restructure** An operation to regenerate or rebuild the database index and, in some cases, data files.

**roll-up** See consolidation.

**root member** The highest member in a dimension branch.

**sampling** The process of selecting a representative portion of an entity to determine the entity's characteristics. See also metadata sampling.

**schema** In relational databases, a logical model that represents the data and the relationships between the data.

**scope** The area of data encompassed by any Essbase operation or setting; for example, the area of data affected by a security setting. Most commonly, scope refers to three levels of granularity, where higher levels encompass lower levels. The levels, from highest to lowest: the entire system (Essbase Server), applications on Essbase Server, or databases within Essbase Server applications. See also persistence.

**scrapping** An inspection of a data source to derive the most basic metadata elements from it. Contrast with introspection.

**security platform** A framework enabling Oracle EPM System products to use external authentication and single sign-on.

**serial calculation** The default calculation setting. Divides a calculation pass into tasks and calculates one task at a time.

**shared member** A member that shares storage space with another member of the same name, preventing duplicate calculation of members that occur multiple times in an Essbase outline.

**Shared Services Registry** The part of the Shared Services repository that manages EPM System deployment information for most EPM System products, including installation directories, database settings, computer names, ports, servers, URLs, and dependent service data.

**sibling** A child member at the same generation as another child member and having the same immediate parent. For example, the members Florida and New York are children of East and each other's siblings.

**single sign-on (SSO)** The ability to log on once and then access multiple applications without being prompted again for authentication.

**slicer** In MaxL DML, the section at the end of a query that begins with and includes the keyword WHERE.

**sparse dimension** In block storage databases, a dimension unlikely to contain data for all member combinations when compared to other dimensions. Contrast with dense dimension. For example, not all customers have data for all products.

**standard dimension** A dimension that is not an attribute dimension.

**stored hierarchy** In aggregate storage databases outlines only, a hierarchy in which the members are aggregated according to the outline structure. Stored hierarchy members have certain restrictions; for example, they cannot contain formulas.

**supervisor** A user with full access to all applications, databases, related files, and security mechanisms for a server.

**suppress rows** A setting that excludes rows containing missing values and underscores characters from spreadsheet reports.

**symmetric multiprocessing (SMP)** A server architecture that enables multiprocessing and multithreading. Performance is not significantly degraded when a large number of users simultaneously connect to an single instance.

**synchronized** The condition that exists when the latest version of a model resides in both the application and in Shared Services. See also model.

**TCP/IP** See Transmission Control Protocol/Internet Protocol.

**text list** In Essbase, an object that stores text values mapped to numeric identifiers. Text Lists enable the use of text measures.

**text measure** In Essbase, a member tagged as Text in the dimension where measures are represented. The cell values are displayed as predefined text. For example, the text measure Satisfaction Index may have the values Low, Medium, and High. See also typed measure, text list, derived text measure.

**time series reporting** A process for reporting data based on a calendar date (for example, year, quarter, month, or week).

**traceability** The ability to track a metadata element to its physical source. For example, in Essbase Studio, a cube schema can be traced from its hierarchies and measure hierarchies to its dimension elements, date/time elements, measures, and, ultimately, to its physical source elements. See also lineage.

**transformation** 1) A process that transforms artifacts so that they function properly in the destination environment after application migration; 2) In data mining, the modification of data (bidirectionally) flowing between the cells in the cube and the algorithm.

**Transmission Control Protocol/Internet Protocol (TCP/IP)** A standard set of communication protocols linking computers with different operating systems and internal architectures. TCP/IP utilities are used to exchange files, send mail, and store data to various computers that are connected to local and wide area networks.

**transparent partition** A shared partition that enables users to access and change data in a remote database as though it is part of a local database.

**triggers** An Essbase feature whereby data is monitored according to user-specified criteria that, when met, cause Essbase to alert the user or system administrator.

**tuple** MDX syntax element that references a cell as an intersection of a member from each dimension. If a dimension is omitted, its top member is implied. Examples: (Jan); (Jan, Sales); ( [Jan], [Sales], [Cola], [Texas], [Actual] ).

**two-pass** An Essbase property that is used to recalculate members that are dependent on the calculated values of other members. Two-pass members are calculated during a second pass through the outline.

**typed measure** In Essbase, a member tagged as Text or Date in the dimension where measures are represented. The cell values are displayed as predefined text or dates.

**unary operator** A mathematical indicator (+, -, \*, /, %) associated with an outline member. The unary operator defines how the member is calculated during a database roll-up.

**Unicode-mode application** An Essbase application wherein character text is encoded in UTF-8, enabling users with computers set up for different languages to share application data.

**unique member name** A nonshared member name that exists only once in a database outline.

**unique member outline** A database outline that is not enabled for duplicate member names.

**upper-level block** A type of data block wherein at least one of the sparse members is a parent-level member.

**user-defined attribute (UDA)** An attribute, associated with members of an outline to describe a characteristic of the members, that can be used to return lists of members that have the specified associated UDA.

**validation** The process of checking a business rule, report script, or partition definition against the outline to ensure that the object being checked is valid.

**varying attribute** An attribute association that changes over one or more dimensions. It can be used to track a value in relation to these dimensions; for example, the varying attribute Sales Representative, associated with the Product dimension, can be used to track the value Customer Sales of several different sales representatives in relation to the Time dimension. Varying attributes can also be used for member selection, such as finding the Products that a Sales Representative was responsible for in May.

**visual cue** A formatted style, such as a font or a color, that highlights specific data value types. Data values may be dimension members; parent, child, or shared members; dynamic calculations; members containing a formula; read-only data cells; read-and-write data cells; or linked objects.

**WITH section** In MaxL DML, an optional section of the query used for creating reusable logic to define sets or members. Sets or custom members can be defined once in the WITH section and then referenced multiple times during a query.

**workbook** An entire spreadsheet file with many worksheets.

**write-back** The ability for a retrieval client, such as a spreadsheet, to update a database value.

**XOLAP** An Essbase multidimensional database that stores only the outline metadata and retrieves all data from a relational database at query time. XOLAP supports aggregate storage databases and applications that contain duplicate member names.



---

# Index

---

## A

AbstractCommandListener class, [32](#)  
AbstractCommandListener.getCommands method, [33](#)  
accessing  
    client plug-ins, [11](#)  
ActivateButton control, [27](#)  
adapter field, [24](#)  
addAll method, [41](#)  
adding  
    a branch to the Enterprise Tree, [14](#)  
    children to tree nodes, [16](#)  
    console tree menu items, [21](#)  
    context menu items to tree nodes, [17](#)  
    internal frame menu items, [21](#)  
    items to menus, [19](#)  
    items to the File > New menu, [18](#)  
    static menu items, [20](#)  
Administration Server  
    described, [7](#)  
Administration Services  
    described, [7](#)  
    Java packages, [10](#)  
    logging level, [43](#)  
Administration Services Console  
    adding a branch to the tree, [14](#)  
    adding functionality, [14](#)  
    class packages, [11](#)  
    described, [7](#)  
    locating plug-ins, [13](#)  
    retrieving the CSS token, [28](#)  
    services, [28](#)  
    writing plug-ins for, [11](#)  
APP\_SCOPE, [41](#)  
ApplyButton control, [27](#)  
AppManCommandListener class, [32](#), [33](#)  
architecture, [8](#)

## B

BackButton control, [27](#)  
Boolean hidden field, [35](#)  
boolean required field, [34](#)  
BooleanComboBox control, [27](#)  
BUILDER\_SCOPE, [41](#)  
ButtonPanel control, [27](#)  
buttons  
    standard, [27](#)  
buttons field, [24](#)

## C

cancelBtn field, [24](#)  
CancelButton control, [27](#)  
children  
    adding to tree nodes, [16](#)  
    permitting plug-ins to add to tree nodes, [17](#)  
Class ClassType field, [34](#)  
class hierarchy  
    for command listeners, [32](#)  
class packages  
    Administration Services Console, [11](#)  
client  
    adding functionality, [14](#)  
    class packages, [11](#)  
    locating plug-ins, [13](#)  
    sending results to, [40](#)  
    writing plug-ins for, [11](#)  
client plug-ins  
    access point, [11](#)  
client tier, [7](#)  
CloseButton control, [27](#)  
code  
    compiling, [42](#)  
    loading, [43](#)  
    packaging, [42](#)  
code samples

- about, [10](#)
- com.essbase.eas.client.intf, [11](#)
- com.essbase.eas.client.manager, [12](#)
- com.essbase.eas.client.plugins, [12](#)
- com.essbase.eas.framework.client.defs.command, [12](#)
- com.essbase.eas.framework.client.defs.login, [12](#)
- com.essbase.eas.framework.client.ui.filedlg, [12](#)
- com.essbase.eas.framework.defs, [12](#)
- com.essbase.eas.i18n, [12](#)
- com.essbase.eas.i18n package, [29](#)
- com.essbase.eas.ui, [12](#)
- com.essbase.eas.ui.ctable, [12](#)
- com.essbase.eas.ui.ctree, [12](#)
- com.essbase.eas.ui.editor, [12](#)
- com.essbase.eas.ui.email, [12](#)
- com.essbase.eas.ui.font, [12](#)
- com.essbase.eas.ui.print, [12](#)
- com.essbase.eas.ui.ptable, [12](#)
- com.essbase.eas.ui.ptree, [12](#)
- com.essbase.eas.ui.tree, [12](#)
- com.essbase.eas.utils, [12](#)
- com.essbase.eas.utils.print, [12](#)
- com.MyPlugin.MiscellaneousHandler, [13](#)
- command arguments
  - grabbing, [40](#)
- command handling methods
  - described, [38](#)
- command listener
  - class hierarchy, [32](#)
- command listeners
  - defined, [32](#)
  - writing, [31](#)
- CommandArgument class, [34](#)
- CommandArgument object, [35](#)
- CommandDescriptor class, [35](#)
- CommandDescriptor objects, [34](#)
- commands, registering, [34](#)
- CommandStatus class, [40](#)
- CommandString class, [34](#)
- CONFIG\_SCOPE, [41](#)
- Configure Plugin Components dialog box, [13](#)
- console tree menu items, adding, [21](#)
- constructors for the StandardDialog class, [24](#)
- context menu items
  - adding to tree nodes, [17](#)
- controls
  - setting focus order, [26](#)

- standard, [27](#)
- CSS token
  - retrieving from the Console, [28](#)

## D

- data
  - storing temporary using the framework, [41](#)
- DefaultScopeTypes, [41](#)
- dialog results, [26](#)
- Dialog.show method, [26](#)
- DialogResult class, [26](#)
- dialogResult field, [24](#)
- DialogUtils.setFocusOrder method, [26](#)
- dispose method, [27](#)
- DoneButton control, [27](#)

## E

- e-mail
  - support for sending, [28](#)
- eas\_client.jar file, [11](#)
- eas\_common.jar file, [11](#)
- Enterprise Tree
  - adding a branch, [14](#)
- essbase\_client.jar file, [11](#)
- essbase\_common.jar file, [11](#)
- Enterprise Tree
  - adding a branch, [14](#)
- essbase\_client.jar file, [11](#)
- essbase\_common.jar file, [11](#)
- EssbaseCommandListener, [40](#)
- EssbaseCommandListener class, [32](#), [33](#)
- EssbaseCommandListener.handleEventPrep method, [33](#)
- example classes, [10](#)
- example code
  - about, [10](#)
- example.java sample code, [35](#)
- ExampleCommandListener class, [37](#)
- exampleCommandListener.java sample code, [38](#)
- exampleCommandString.java sample code, [36](#)
- exampleDescriptor.java sample code, [37](#)
- extending Administration Services Console, [11](#)

## F

- File > New menu
  - adding items to, [18](#)
- FinishButton control, [27](#)
- focus order of controls, setting, [26](#)
- framework
  - using to store temporary data, [41](#)

framework\_client.jar file, [11](#), [42](#)  
 framework\_common.jar file, [11](#), [42](#)  
 framework\_server.jar file, [42](#)  
 functionality  
   adding to Administration Services Console, [14](#)

**G**

get method, [42](#)  
 getClassType method, [34](#)  
 getCommands method, [34](#), [38](#)  
 getContextMenuItemsFor method, [17](#)  
 getDefaultValue method, [35](#)  
 getName method, [34](#)  
 getObjectsToEmail method, [28](#)  
 getTreeNodeChildren method, [16](#)  
 grabbing command arguments, [40](#)

**H**

handleCancel method, [26](#), [27](#)  
 handleEvenPost method, [33](#)  
 handleEvent method, [33](#)  
 handleEventException method, [33](#)  
 handleEventPrep method, [33](#), [40](#)  
 handleHelp method, [26](#)  
 handleOk method, [19](#), [26](#), [27](#)  
 handleWindowClosed method, [27](#)  
 handleWindowClosing method, [27](#)  
 handleWindowOpened method, [27](#)  
 helpBtn field, [24](#)  
 HelpButton control, [27](#)

**I**

internal frame menu items, [21](#)  
 InternalFrame class, [28](#)  
 internationalization utilities, [29](#)  
 isEmailable method, [28](#)  
 isHidden method, [35](#)  
 isRequired method, [34](#)

**J**

Java Introspection, [9](#)  
 Java packages  
   for Administration Services, [10](#)  
 Java plug-in components  
   described, [8](#)

  requirements for using, [9](#)  
 Java plug-ins  
   packaging, [29](#)  
 Java Swing, [9](#)

**L**

ListMoverPanel control, [27](#)  
 loading code, [43](#)  
 localization utilities, [29](#)  
 logging level, [43](#)

**M**

manifest file, [43](#)  
 menu items  
   adding, [19](#)  
   adding internal frame, [21](#)  
   adding to tree nodes, [17](#)  
   console tree, [21](#)  
 menus  
   adding items to, [19](#)  
 method signatures, [39](#)  
 methods  
   command handling, [38](#)  
 middle tier, [7](#)  
 MiscellaneousHandler.class, [13](#)

**N**

name parameter, [32](#)  
 NextButton control, [27](#)  
 nodes  
   adding children, [16](#)  
   adding context menu items to, [17](#)  
   permitting plug-ins to add children to, [17](#)  
 NumericTextField control, [27](#)

**O**

Object defaultValue field, [35](#)  
 okBtn field, [23](#)  
 OkButton control, [27](#)  
 op parameter, [32](#)

**P**

packaging plug-ins, [29](#)  
 packaging the code, [42](#)  
 password parameter, [32](#)

permitting plug-ins to add children to tree nodes, [17](#)

plug-ins

access point for client, [11](#)

how the client locates, [13](#)

packaging, [29](#)

writing client, [11](#)

populatePanel method, [18](#)

populateTree method, [14](#)

public classes

Administration Services Console, [11](#)

## R

ReadOnlyTextFrame control, [27](#)

RefreshButton control, [27](#)

registering commands, [34](#)

remove method, [42](#)

REQUEST\_SCOPE, [42](#)

requirements

for using Java plug-in components, [9](#)

ResetButton control, [27](#)

resources field, [24](#)

## S

sample code

about, [10](#)

example.java, [35](#)

exampleCommandListener.java, [38](#)

exampleCommandString.java, [36](#)

exampleDescriptor.java, [37](#)

Save As, handling, [21](#)

SaveAsRequestor interface, [21](#)

saveDialogBounds field, [24](#)

sending e-mail, [28](#)

sending results back to the client, [40](#)

server-side command listeners

writing, [31](#)

services

for Administration Services Console, [28](#)

SESSION\_SCOPE, [41](#)

set method, [42](#)

setting

focus order of controls, [26](#)

SimpleWizardPanel control, [27](#)

standard buttons and controls, [27](#)

standard controls, [23](#)

StandardDialog class, [23](#)

methods that can be overridden, [26](#)

StandardDialog class constructors, [24](#)

StandardDialog class name, [24](#)

StandardDialog default action, [25](#)

StandardDialog initialization, [24](#)

StandardDialog results, [26](#)

static menu items, adding, [20](#)

StoreService interface, [42](#)

String name field, [34](#)

## T

temporary data

storing using the framework, [41](#)

toString method, [34](#), [37](#)

tree nodes

adding children, [16](#)

adding context menu items to, [17](#)

permitting plug-ins to add children to, [17](#)

## U

USER\_SCOPE, [42](#)

utilities for localization, [29](#)

utility classes, [44](#)

## V

validateSession method, [40](#)

VerticalPairPanel control, [27](#)

## W

war file, [43](#)

WizardPanel control, [27](#)

writing server-side command listeners, [31](#)