Telewindows2 Toolkit

Java Developer's Guide

Application Classes

Version 1.2 Rev. 2

gensym

Telewindows2 Toolkit Java Developer's Guide, Version 1.2 Rev. 2
May 2002

Gensym Corporation
52 Second Avenue
Burlington, MA 01803 USA
Telephone: (781) 265-7100
Fax: (781) 272-7101

Part Number: DOC067-122

Contents Summary

Contents

Preface

Describes this document and the conventions that it uses.

Using this Guide

This guide uses a bottom-up organizational approach to describe how to build client user interface applications for G2, using Java. A bottom-up approach means:

- The chapters progress sequentially from describing how to integrate individual, generic, UI components, and G2 application-specific components in any Java application, to describing how to build complete applications.
- Later chapters assume knowledge of earlier chapters.

The exception to this approach is Chapter 2, "Guided Tour of the Telewindows2 Toolkit Shell" on page 33, which provides a walk-through of the default application shell user interface.

The following table summarizes the topics covered in each chapter in Parts II and III:

Topic	Example	Chapter
Informational and input dialogs	Dialogs with text boxes, selection dialogs, and error dialogs.	Chapter 4, “Using Standard Dialogs”
Application components for building UI	Menus, toolbars, and commands.	Chapter 5, “Creating Menus and Toolbars”
Palettes	Generic palettes of objects and palettes of G2 objects.	Chapter 6, “Creating Palettes” on page 163
Multiple document interface (MDI) containers and managers	MDI frame, document, manager, and listener.	Chapter 7, “Creating Multiple Document Interface Containers”
MDI document types	MDI documents that display views of G2 data, with context-specific menu bars.	Chapter 8, “Using Telewindows2 Toolkit MDI Documents”
Application foundation classes that handle connections to G2 and manage the application frame	Single document interface (SDI), multiple document interface (MDI), and generic UI applications.	Chapter 9, “Creating Telewindows2 Toolkit Applications”
G2 application-specific dialogs and UI components	A dialog for logging into a secure G2, and a panel for switching the current connection.	Chapter 10, “Using Shell Dialogs and UI Controls” on page 259
G2 application-specific commands	Commands for connecting to G2 and getting a named workspace.	Chapter 11, “Using Shell Commands”
Source code for the TW2 Toolkit Java application shell.	A multiple connection MDI application.	Chapter 12, “Understanding the Telewindows2 Toolkit Shell”

For detailed road maps of topics covered in this guide, see Chapter 3, “Road Maps to Using This Guide” on page 61.

Audience

This guide is for user interface developers, who use TW2 Toolkit application classes to build G2 client applications in Java.

In general, this guide stands on its own to describe how UI developers can use TW2 Toolkit application classes to build a G2 client UI. Using these classes alone, you can create a simple user interface that provides an application frame, menus and toolbars, basic commands for interacting with the KB, and basic connectivity to a G2 server. If your KB provides navigation through GUIDE/UIL buttons, you can use these buttons for navigation in the client, as well.

If your application needs to customize the way in which TW2 Toolkit application classes handle connections, display and manipulate workspace views, or manage and launch dialogs, you will also need to refer to the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*. This guide references the components and core classes guide, where relevant.

A Note About the API

The techniques by which Telewindows2 Toolkit implements its capabilities are subject to change at any time without notice or explanation, and are expected to change as the toolkit evolves. These techniques, and any changes to them, will not be described in any documentation.

Therefore, it is essential that you use TW2 Toolkit exclusively through its API as described here and in the API documentation. Any methods or classes that are not included in the API are subject to change without notice. Any code that calls undocumented methods may cease to work in newer versions.

Conventions

Typographic

Convention Examples	Description
g2-window, g2-window-1, gfr-top-level, sys-mod	G2 class names, instance names, workspace names, and module names
history-keeping-spec, temperature	G2 attribute names
true, 1.234, ok, “Burlington, MA”	Attribute values and values specified or viewed through dialogs

Convention Examples	Description
Main Menu > Start KB Workspace > New Object create subworkspace Start Procedure	G2 menu choices and button labels
conclude that the x of y ...	Text of G2 procedures, methods, functions, formulas, and expressions
new-argument	User-specified values in syntax descriptions
<u>*text-string*</u>	Return values of G2 procedures and methods in syntax descriptions
File Name, OK, Apply, Cancel, General, Edit Scroll Area	GUIDE and native dialog fields, button labels, tabs, and titles
File > Save Properties	GMS and native top-level menu choices and native popup menu choices
workspace	Glossary terms
`c:\Program Files\Gensym\g2`	Windows pathnames
`/usr/gensym/g2/kbs`	UNIX pathnames
`spreadsh.kb`	File names
`g2 -kb top.kb`	Operating system commands
`public void main()` `gsi_start`	Java, C and all other external code

Note Syntax conventions are fully described in the *G2 Reference Manual*.

Procedure Signatures

A procedure signature is a complete syntactic summary of a procedure or method. A procedure signature shows values supplied by the user in *italics*, and the value (if any) returned by the procedure *underlined*. Each value is followed by its type:

g2-clone-and-transfer-objects
(*list*: class item-list, *to-workspace*: class kb-workspace,
delta-x: integer, *delta-y*: integer)
–> *transferred-items*: g2-list

Related Documentation

Telewindows2 Toolkit

Online Files

The following document is available in the following directory, depending on your platform:

NT: `%SEQUOIA_HOME%\readme-tw2.html`

UNIX: `$SEQUOIA_HOME/readme-tw2.html`

Java Developer's Guides

The online files are located in this directory, by default, depending on your platform:

NT: `c:\Program Files\Gensym\g2-6.1\doc\tw2\Java\docs\guides\`

UNIX: `/usr/gensym/g2-6.1/doc/tw2/Java/docs/guides/`

- *Telewindows2 Toolkit Release Notes*
- *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*
- *Telewindows2 Toolkit Java Developer's Guide: Application Classes*
- *Telewindows2 Toolkit Java Demos Guide*
- *BeanXporter User's Guide*

G2 JavaLink

Online Files

The following document is available in the following directory, depending on your platform:

NT: `%JAVALINK_HOME%\readme-javalink.html`

UNIX: `$JAVALINK_HOME/readme-javalink.html`

User's Guides

The online files are located in this directory, depending on your platform:

NT: `c:\Program Files\Gensym\g2-6.1\doc\javalink\docs\guides\`

UNIX: `/usr/gensym/g2-6.1/doc/javalink/docs/guides/`

- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

Java Reference Material

- JDK 1.3 documentation set *
- *The Java Language Specification* (Gosling, Joy, Steele. Addison Wesley) *
- *The Java Bean Specification V1.0* *
- *These and other Java documents can be downloaded from Sun Microsystems' Java web site at `http://www.javasoft.com`.

G2 Core Technology

- *G2 Bundle Release Notes*
- *Getting Started with G2 Tutorials*
- *G2 Reference Manual, Volumes I and II*
- *G2 Developer's Guide*
- *G2 System Procedures Reference Manual*
- *G2 Class Reference Manual*
- *Telewindows User's Guide*
- *G2 Gateway Bridge Developer's Guide*

G2 Utilities

- *G2 ProTools User's Guide*
- *G2 Foundation Resources User's Guide*
- *G2 Developer's Interface User's Guide*
- *G2 Menu System User's Guide*
- *G2 XL Spreadsheet User's Guide*
- *G2 Dynamic Displays User's Guide*
- *G2 GUIDE User's Guide*
- *G2 GUIDE/UIL Procedures Reference Manual*
- *G2 OnLine Documentation Developer's Guide*
- *G2 OnLine Documentation User's Guide*

G2 Diagnostic Assistant

- *GDA User's Guide*
- *GDA Reference Manual*
- *GDA API Reference*

Bridges and External Systems

- *G2 WebLink User's Guide*
- *G2 ActiveXLink User's Guide*
- *G2 CORBALink User's Guide*

- *G2 OPCLink User's Guide*
- *G2-Oracle Bridge Release Notes*
- *G2-Sybase Bridge Release Notes*
- *G2-ODBC Bridge Release Notes*
- *G2 Database Bridge User's Guide*

Customer Support Services

You can obtain help with this or any Gensym product from Gensym Customer Support. Help is available online, by telephone, by fax, and by email.

To obtain customer support online:

➔ Access G2 HelpLink at `http://www.gensym-support.com`.

You will be asked to log in to an existing account or create a new account if necessary. G2 HelpLink allows you to:

- Register your question with Customer Support by creating an Issue.
- Query, link to, and review existing issues.
- Share issues with other users in your group.
- Query for Bugs, Suggestions, and Resolutions.

To obtain customer support by telephone, fax, or email:

➔ Use the following numbers and addresses:

	Americas	Europe, Middle-East, Africa (EMEA)
Phone	(781) 265-7301	+31-71-5682622
Fax	(781) 265-7255	+31-71-5682621
Email	`service@gensym.com`	`service-ema@gensym.com`

Part I

Introduction

1

Overview

Provides an overview of the Telewindows2 Toolkit packages, Java requirements, and application features you can use to build a G2 client user interface application, using Telewindows2 Toolkit.

Introduction

The *Telewindows2 Toolkit Java Developer's Guide: Application Classes* is for Java application developers who want to build native, client applications that provide user interfaces for viewing and manipulating data in a G2 server.

Here is what each of these phrases means in more detail for the Telewindows2 (TW2) Toolkit:

Java application developer	A programmer who builds applications in a Java programming environment, such as Symantec Visual Café, IBM's Visual Age, or pure Java.
Native	Conforms to the native "look-and-feel" of the window system on which the UI application is running.
Client application	An application that runs on any platform and interacts through a network connection with a server.
User interface	Any kind of visual application that allows end users or developers to interact with data, using menus, toolbars, and dialogs.
View data	To display a visual representation of any type of G2 data, such as a KB workspace or item properties dialog.
Manipulate data	To modify data in the G2 server through a native, client user interface.
G2 server	A running G2 executable, which is the source of all G2 data that users view and manipulate.

For additional terms relating to the TW2 Toolkit, see the "Glossary" on page 339.

Note This guide does not describe any Java terms or concepts, which are explained fully in numerous, readily available Java programming language books, as well as on Sun's Java website at `www.java.sun.com`.

Packages

To build a user interface application for interacting with G2, using Telewindows2 (TW2) Toolkit, you use classes in these categories of packages:

- Application packages — Provide stand-alone classes for use in any Java application.
- Shell packages — Provide classes that you can use to build G2 client applications, and the source code for the TW2 Toolkit default application shell.

For a listing of all packages, see the API documentation in this location in your G2 Bundle product directory:

NT:`\doc\tw2\Java\Docs\api\packages.html`

UNIX:`/doc/tw2/Java/Docs/api/packages.html`

Package Categories

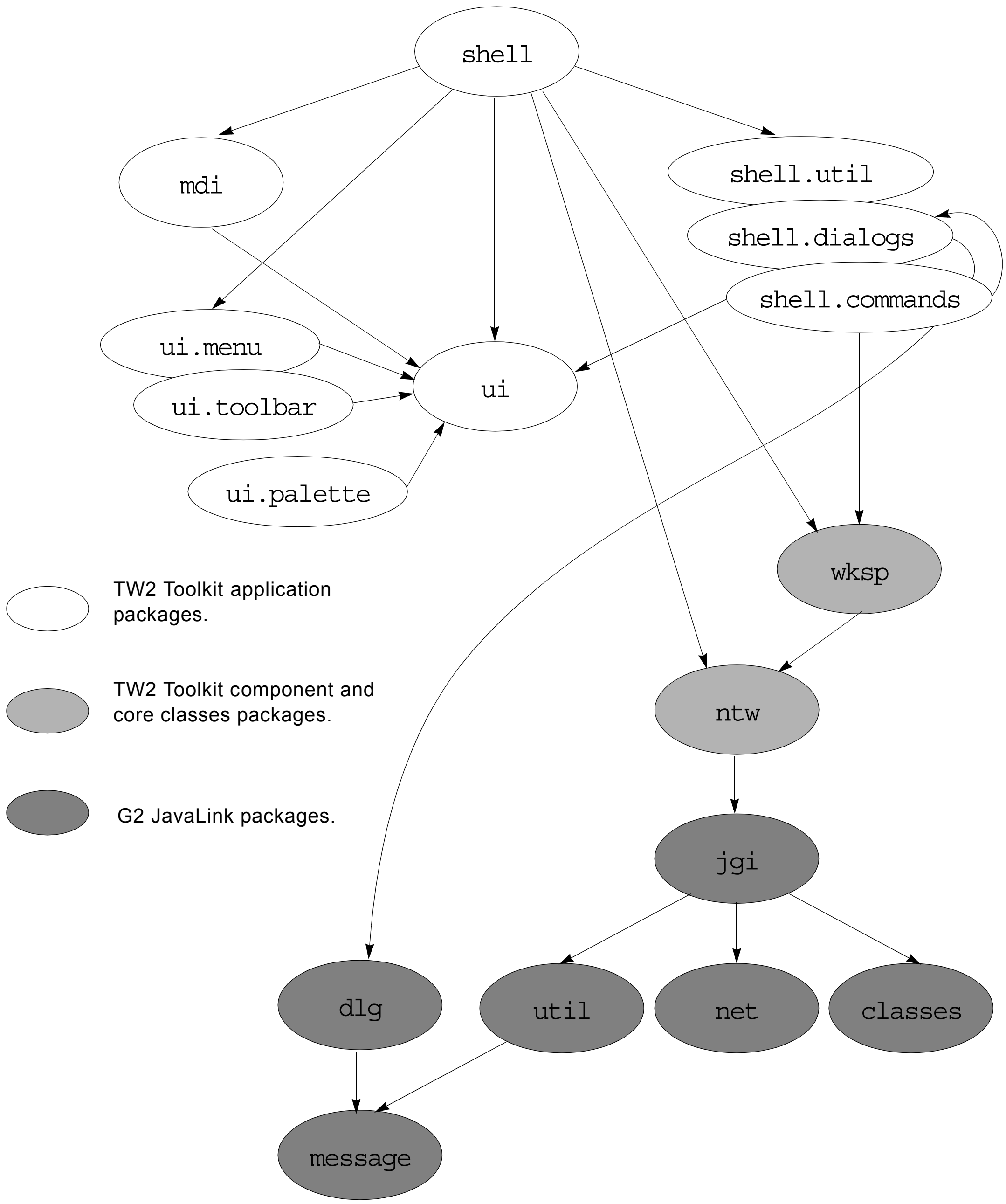
This table describes the contents of the available packages in each category:

Package	Description
Application Packages	
`com.gensym.dlg`	Standard dialogs that display information to the user and get input from the user.
`com.gensym.ui`	Basic interfaces and classes to support menus, toolbars, and palettes.
`com.gensym.ui.menu` `com.gensym.ui.toolbar`	User interface containers that display and represent user actions and listen for associated events.
`com.gensym.ui.palette`	Generic classes to support palettes.
`com.gensym.mdi`	Multiple document interface (MDI) frames, and their associated containers and managers.

Package	Description
Shell Packages	
`com.gensym.shell.dialogs`	Application-specific dialogs and UI controls that support common interactions with a G2 server.
`com.gensym.shell.commands`	Common user interactions with a G2 server, which you can include in a menu or toolbar.
`com.gensym.shell.util`	Support for: • Managing multiple connections to G2 and handling associated events. • Creating single and multiple document applications that manage application frames and connections to G2. • MDI document types that display workspace views.

Package Dependencies

This diagram illustrates the package dependencies of the application and shell packages on other TW2 Toolkit packages and G2 JavaLink packages:



Supporting Features

The underlying features of Telewindows2 Toolkit that allow users to view and manipulate G2 server data through a native, client user interface are:

- **Telewindow2 Toolkit components and core classes** — Provide the basic support for connecting to a G2 server, displaying and manipulating data through a workspace view, and handling the associated events.
- **G2 JavaLink** — Provides the underlying technology that enables TW2 Toolkit components to access and manipulate data in a G2 server, and to represent G2 items as components.

For additional information on building G2 client applications, using TW2 Toolkit components and core classes, and G2 JavaLink, see these guides:

- *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*
- *G2 JavaLink User's Guide*
- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

Java Requirements

To use Telewindows2 Toolkit to build G2 client applications in Java, you must have a working knowledge of:

- **Java programming** — Creating Java applications that:
  - Manipulate the properties, events, and methods of classes and interfaces.
  - Use the Java 1.1. event model.
  - Support internationalization.
- **Java Abstract Windowing Toolkit (AWT) and Java Foundation Classes (JFC)** — The base classes upon which the TW2 Toolkit user interface classes are built.
- **User interface development** — The general technique of constructing a user interface by:
  - Adding Java components to containers.
  - Arranging those components, using layout managers.

For more information, see the `java.sun.com` website.

Telewindows2 Toolkit Application Classes

Telewindows2 Toolkit provides several categories of classes, which you can use to build client user interface applications for G2:

- **Graphical user interface classes**, which let you create:
  - Standard informational dialogs and dialogs that interact with G2 items, such as error dialogs, input dialogs, and selection dialogs.
  - Menus and toolbars.
  - Palettes for cloning G2 items onto a KB workspace.
  - Multiple document interface (MDI) applications, which includes MDI document types for displaying views of G2 server data, such as workspace views.
- **Application foundation classes**, which let you create these types of applications that manage connections to G2:
  - Generic UI applications.
  - Single document interface (SDI) applications.
  - Multiple document interface (MDI) applications.
- **Shell dialogs and UI controls**, which allow you to perform common interactions with G2, such as logging in and switching the user mode.
- **Shell commands**, which provide common user interactions with G2 for inclusion in menus and toolbars, such as making a connection, changing the G2 run state, and creating and getting a KB workspace.
- **Shell classes**, which provide the classes that define the TW2 Toolkit default application shell, a simple user interface for connecting to multiple G2s, and displaying and manipulating G2 items through a workspace view.

The following sections show examples of some of these features and provide references to the chapters in this guide where the feature is discussed.

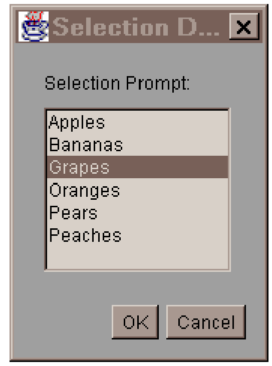
Standard Dialogs

Telewindows2 Toolkit provides a number of standard dialogs that you can use directly in your application. These dialog classes are part of G2 JavaLink.

You use the dialogs to provide information to and obtain input from the user. The dialogs provide standard buttons for dismissing and cancelling the dialog, as well as icons appropriate to the particular type of dialog.

This figure shows examples of some of the dialogs in the `com.gensym.dlg` package:

`ErrorDialog`



`SelectionDialog`

`InputDialog`

You can customize the buttons and icon for any standard dialog by subclassing the dialog. You can also create standard dialogs with different types of controls, where you specify the layout of the controls.

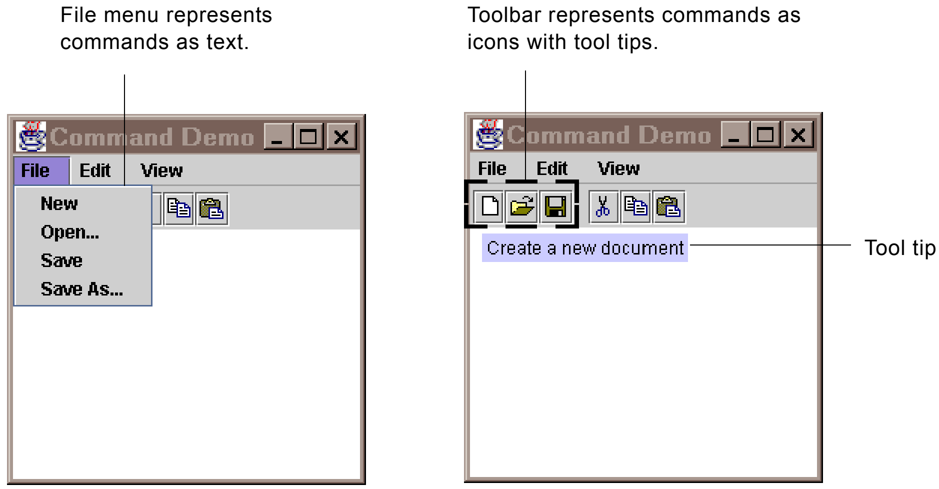
For detailed information on creating standard dialogs, see Chapter 4, “Using Standard Dialogs” on page 71.

Menus and Toolbars

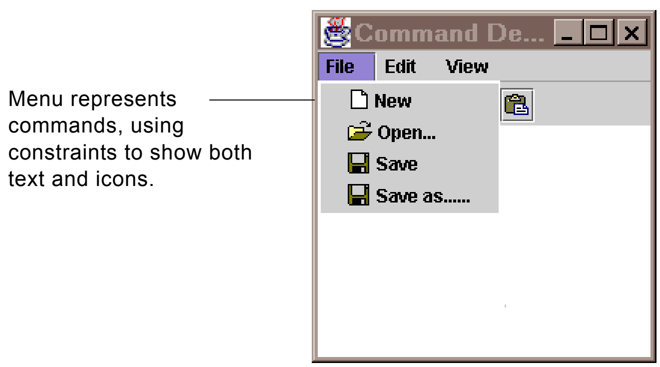
Telewindows2 Toolkit provides a number of classes and interfaces to support “commands,” which are actions that the user can perform through the UI, and “command-aware containers,” which are containers that know how to represent those commands, such as menus and toolbars. These TW2 Toolkit classes support:

- Encapsulation by keeping the command separate from the UI that represents it.
- Reusability by allowing you to add the same command to more than one command-aware container.
- Command availability by notifying command-aware containers of changes in command state.

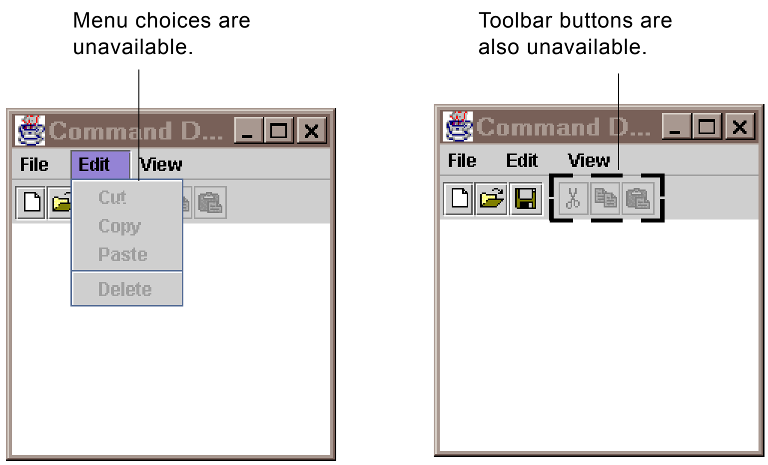
For example, this figure shows a representation of the same command in a menu and in a toolbar:



This figure shows an application with a menu that includes both text and icons:



This figure shows the Edit menu of an application where neither the Cut, Copy, and Paste menu choice nor the corresponding toolbar button is available, because no item is currently selected:



You create commands by extending one of these two classes in the `com.gensym.ui` package:

- `AbstractCommand` — Creates a set of related actions.
- `AbstractStructuredCommand` — Creates a set of related actions with a hierarchical structure or logical grouping.

These classes provide default implementations of the `Command` interface. You can implement this interface to customize the way in which a command handles event notification or the behavior of its abstract methods.

TW2 Toolkit provides these classes for creating command-aware containers:

Class	Type of Container
`CMenu`	Pulldown menu
`ToolBar`	Toolbar
`CMenuBar`	Menu bar
`CPopupMenu`	Popup menu

The classes located in the `com.gensym.ui.menu` and `com.gensym.ui.toolbar` package inherit from classes in the `javax.swing` package. The classes in the `com.gensym.ui.menu.awt` package inherit from classes in the `java.awt` package.

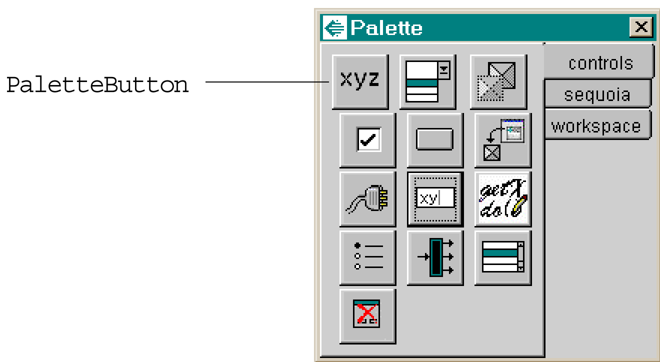
For detailed information on creating commands and adding them to command-aware containers, see Chapter 5, “Creating Menus and Toolbars” on page 113.

Palettes

Telewindows2 Toolkit provides a number of classes in different packages for creating palettes of objects. You can create these types of palettes, where each item is represented as a palette button:

- A generic palette of items.
- A generic palette of items with a hierarchical structure.
- A palette of G2 items.

For example, when you load the `sequoia.jar` file into a Java visual programming environment, you see a generic palette such as the following, which consists of a group of palette buttons for creating dialog controls and buttons for switching the palette:



TW2 Toolkit provides these classes in these packages for creating palettes and palette buttons:

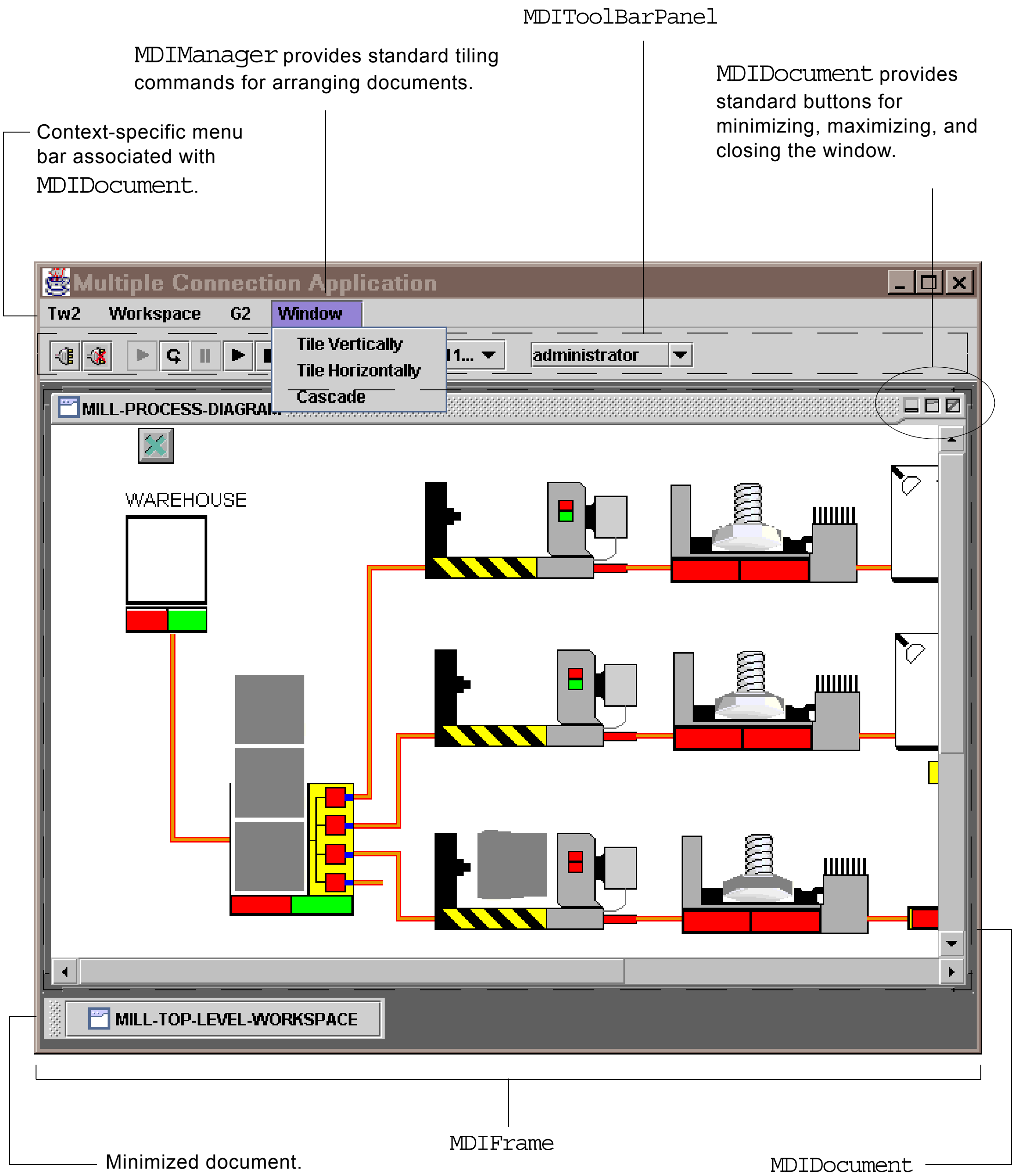
- These classes are located in the `com.gensym.ui.palette` package:
  - `Palette` and `PaletteButton` — Create a generic palette and associated buttons.
  - `PaletteListener` — Receives notification when a palette is created.
  - `PaletteDropTarget` — An interface that you implement to receive notification when a `PaletteButton` gets toggled.
- These classes are located in the `com.gensym.ui` package:
  - `ObjectCreator` — An interface that creates a set of `PaletteButton` objects for each object that gets created.
  - `ObjectCreatorListener` — Receives notification when the availability, icon, or description of any `ObjectCreator` changes.
  - `StructuredObjectCreator` — Creates a hierarchical structure of `PaletteButton` objects.

  - `StructuredObjectCreatorListener` — Receives notification when the structure of a `StructuredObjectCreator` changes.
  - `ObjectFactory` — Determines the type of object a `PaletteButton` creates.
- These classes are located in the `com.gensym.ntw.util` package:
  - `G2Palette` — Creates a `PaletteButton` from a G2 class.
  - `G2ObjectCreator` — A default implementation of the `StructuredObjectCreator` interface that creates a hierarchical structure of `PaletteButton` objects from a hierarchical set of G2 class.
- This class is located in the `com.gensym.clscupgr.gfr` package:
  - `GFRPalette` — Creates a palette from a G2 Foundation Resources (GFR) palette.

For detailed information on creating generic palettes, palettes of G2 objects, and GFR palettes, see Chapter 6, "Creating Palettes" on page 163.

Multiple Document Interface Containers

The Telewindows2 Toolkit default application shell is an example of an MDI application that provides the following MDI containers and features:



Theses classes in the `com.gensym.mdi` package allow you to create multiple document interface (MDI) applications:

- `MDIFrame` — A multiple document interface frame that contains child frames, or `MDIDocuments`.
- `MDIDocument` — A child frame within an `MDIFrame` that:
  - Displays views into your application's data.
  - Provides context-specific menu bars and toolbars.
  - Contains standard buttons for minimizing, maximizing, and closing the document window.
- `MDIToolBarPanel` — A container for displaying one or more toolbars in an `MDIFrame`.
- `MDIManager` — Provides support for:
  - Adding documents to the frame and swapping in context-specific menu bars and toolbars.
  - Getting the active document or an array of open documents.
  - Activating a particular document.
  - Providing a Window menu with commands for arranging multiple documents within the application frame.
  - Handling event notification when a new document is added to the frame.
- `MDIEvent` and `MDIListener` — The event is delivered by the manager when a document is added to the frame, and the listener receives notification of those events.
- `MDITilingConstants` — An interface that provides constants for use when getting standard tiling commands for arranging documents vertically, horizontally, or in a cascade within the frame.

For detailed information on MDI containers and managers, see Chapter 7, "Creating Multiple Document Interface Containers" on page 187.

Telewindows2 Toolkit MDI Documents

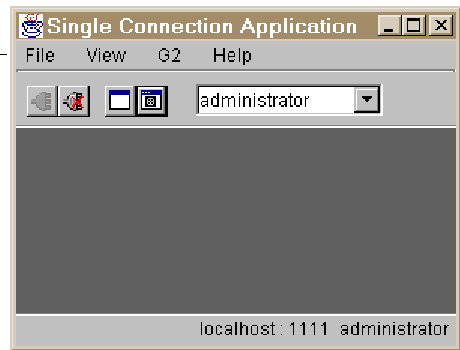
Telewindows2 Toolkit provides two `MDIDocument` types in the `com.gensym.shell.util` package that contain views of G2 server data:

- `TW2Document` — A document that you can extend to display any view into the G2 server's data, for example, a class manager.
- `WorkspaceDocument` — A document for displaying a KB workspace, which you can extend to provide a context-specific menu bar and/or toolbars for interacting with the KB workspace.

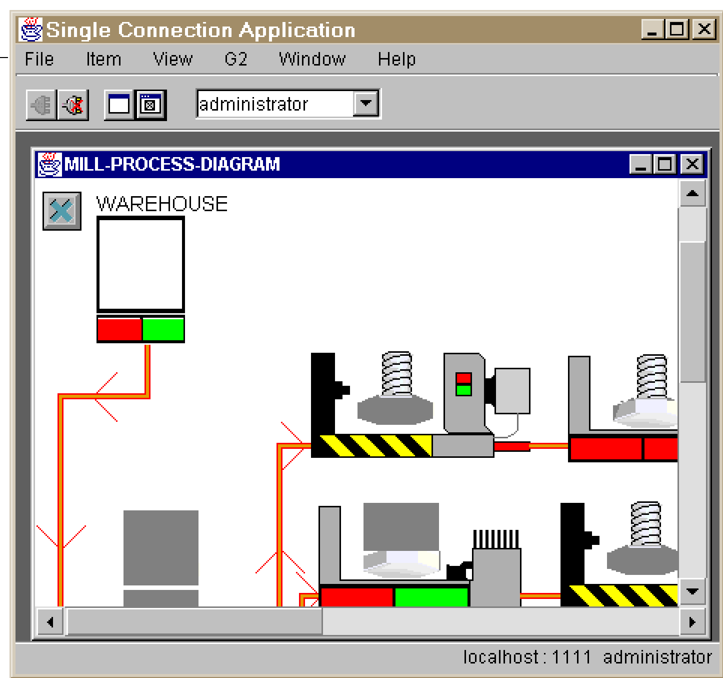
An example of a type of `WorkspaceDocument` is `com.gensym.shell.ShellWorkspaceDocument`, which provides the File, Edit, Item, Workspace, G2, Window, and Help menus in its context-specific menu bar.

This figure shows a single connection application with its default menu bar which appears when no workspace document is visible, and the same application with its context-specific menu bar, which appears when a workspace document has focus:

Default menu bar.



Context-specific menu bar.



For detailed information, see Chapter 8, "Using Telewindows2 Toolkit MDI Documents" on page 207.

Application Foundation Classes

Telewindows2 Toolkit provides foundation classes that you can extend to build the following types of applications:

- **Generic UI applications**, which allow you to interact with G2 through any type of container application.
- **Single document interface (SDI) applications**, which provide a single document frame for displaying G2 server data and which support G2 connections.
- **Multiple document interface (MDI) applications**, which provide multiple document windows within a single frame for displaying G2 server data and which support G2 connections.

The SDI and MDI application foundation classes provide support for connecting to single or multiple G2 servers.

They also provide support for various other required and optional features, such as using command line arguments for connecting to G2 and specifying the frame geometry.

The SDI and MDI application foundation classes are located in the `com.gensym.shell.util` package, while the generic UI application foundation class is in the `com.gensym.core` package, which is part of G2 JavaLink.

The following sections provide examples of each type application.

For detailed information about determining the type of application foundation class to extend and about building TW2 Toolkit applications, see Chapter 9, "Creating Telewindows2 Toolkit Applications" on page 219.

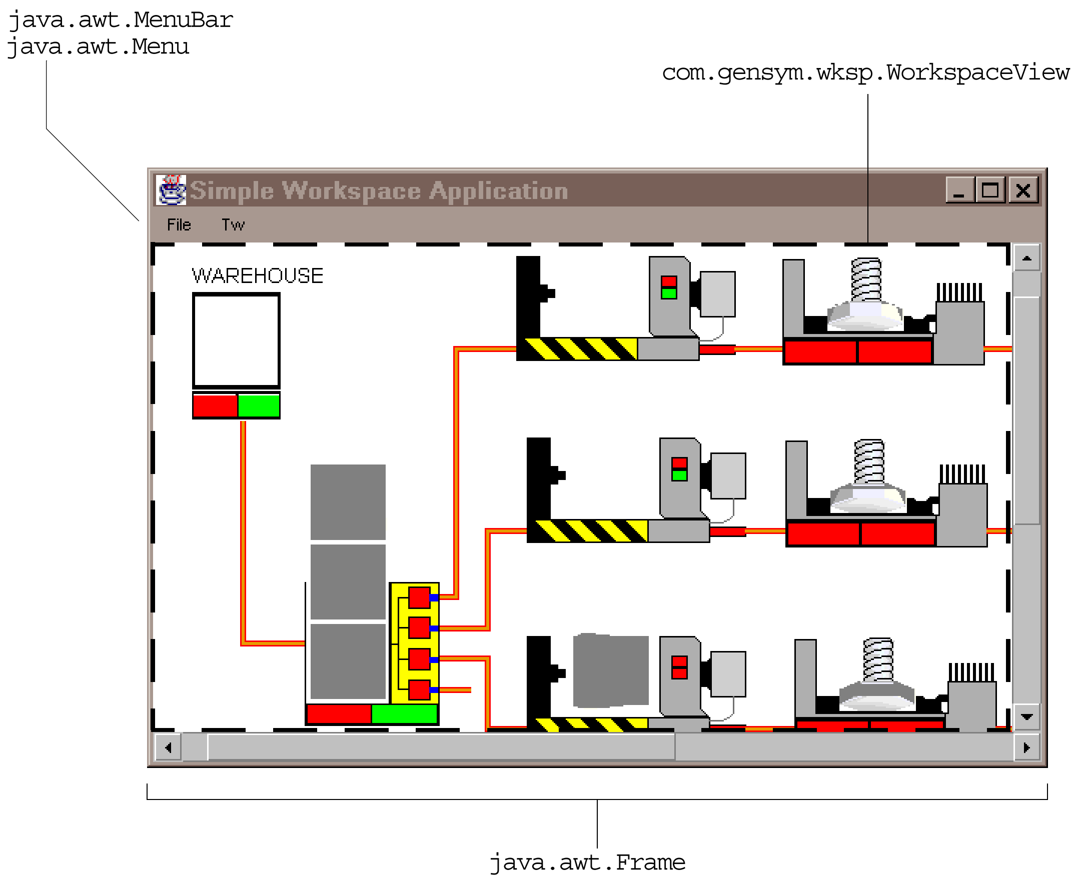
Generic UI Applications

You create a generic UI application by extending `UiApplication`.

You provide the application frame and its UI containers, such as menus and toolbars. You can use Java and/or TW2 Toolkit classes to build the application. For example, the following simple workspace application shows a generic UI application for:

- Opening and closing connections to G2.
- Displaying a named KB workspace.

Note TW2 Toolkit applications allow you to navigate unnamed KB workspaces by using GUIDE/UIL navigation buttons that exist in the KB.



The application uses these classes to implement these features:

- `java.awt` classes to implement the application frame, the menu bar, and the individual menus and menu items.
- `java.awt` events and listeners to handle action events associated with choosing an item from a menu.
- TW2 Toolkit application classes to make the connection to G2 and display a workspace view.

For details about the features of a generic UI application, see “UiApplication” on page 229.

Single Document Interface Applications

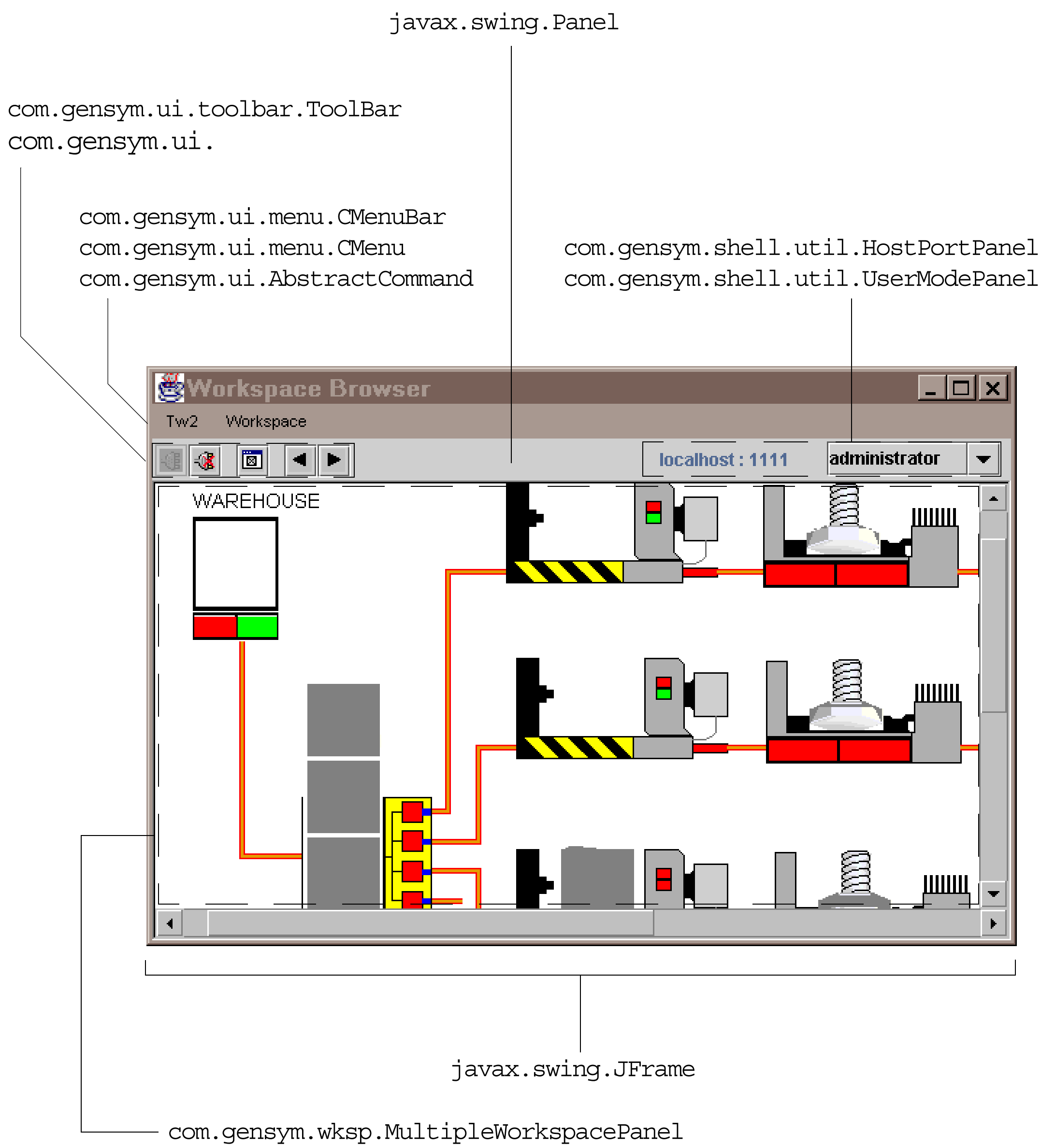
An SDI application provides a frame that contains a single document window in which to display workspace views and other G2 server data.

You create an SDI application by extending `TW2Application`.

The application foundation class provides methods for getting and setting the application frame, and getting and setting the connection. You must provide the UI containers such as menus and toolbars.

For example, the following workspace browser application shows a simple SDI application for:

- Opening and closing connections to G2 by using a menu or toolbar.
- Getting a named workspace by using a menu or toolbar.
- Browsing through multiple named workspaces by clicking the previous and next buttons on the toolbar.
- Displaying the current connection on the toolbar.
- Switching the user mode from the toolbar.



This application uses...	To implement these features...
`javax.swing` components	The application frame, toolbar panel, toolbar layout.
`java.awt` events and listeners	Handle action events associated with choosing an item from a menu or clicking a toolbar button.

This application uses...	To implement these features...
`java.javabeans` events and listeners	Handle events associated with displaying workspaces within the application frame, and cycling through multiple workspaces by using the previous and next buttons in the toolbar.
TW2 Toolkit graphical UI classes	The menu bar, toolbar buttons, and panels for switching the connection and user mode from the toolbar.
TW2 Toolkit graphical UI classes	Handle connections to G2 and represent multiple KB workspaces in a panel.

For details about building SDI applications, see:

- “Creating Telewindows2 Toolkit Applications” on page 233.
- “Creating Single Document Interface Applications” on page 247.

Multiple Document Interface Applications

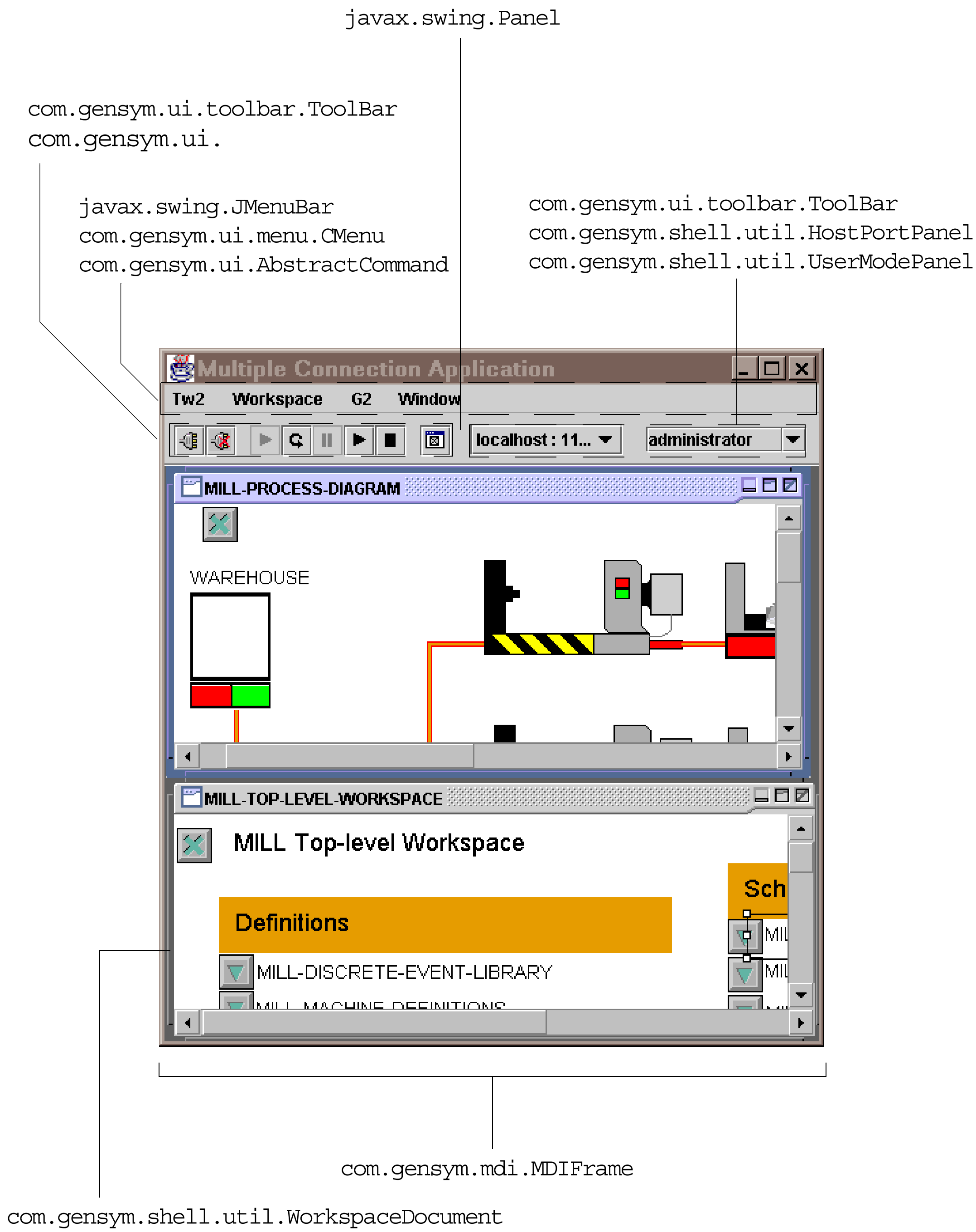
An MDI application provides a frame that contains multiple document windows in which to display workspace views for simultaneous viewing.

You create an MDI application by extending `TW2MDIApplication`.

The application provides methods for getting and setting the application frame, and getting and setting the connection. You must provide the UI containers such as menus and toolbars.

For example, the following multiple connection application shell shows a simple MDI application for:

- Opening and closing multiple connections to G2 by using the menu or toolbar.
- Switching between those connections by choosing the current connection from the toolbar.
- Switching the user mode from the toolbar.
- Displaying multiple named workspaces in different document windows by using the menu or toolbar.
- Arranging the multiple documents windows vertically, horizontally, or in a cascade.
- Controlling the G2 run state by using the menu or toolbar.



Primarily, the application uses TW2 Toolkit application and UI classes to implement these features:

- The MDI frame that displays the multiple windows.

- Multiple workspace document windows and a standard Window menu for arranging them.
- Menus, menu choices, toolbar panel, toolbars, toolbar buttons, and panels for showing the connection and switching the user mode from the toolbar.
- Standard commands for opening connections to G2, switching the G2 run state, getting named workspaces and displaying them in document windows, and exiting the application.
- Multiple connections to G2.
- Event handling associated with switching the current connection to G2, which the application uses to create its own command for disconnecting from G2.
- Events handling associated with programmatically showing a KB workspace in G2.

The only application features that use `javax.swing` are the:

- Menu bar.
- Font, color, and style of the application windows.
- Initial splash image, which the application displays when it is launched.

For details about building MDI applications, see:

- “Creating Telewindows2 Toolkit Applications” on page 233.
- “Creating Multiple Document Interface Applications” on page 251.

Connections to G2

All TW2 Toolkit applications must provide a way of connecting to G2, from the command line and/or through the user interface. TW2 Toolkit applications allow these types of connections to G2:

- Single connections, where the user connects to a single G2 server through an implementation of the `com.gensym.ntw.TwAccess` interface, such as a `TwGateway`.
- Multiple connections, where the user simultaneously connects to multiple G2 servers by using a `ConnectionManager`, which manages those connections.

Thus, if your application is an MDI application that supports multiple connections, the user can display multiple workspace views simultaneously and easily switch between them.

Both SDI and MDI applications allow single or multiple connections.

The `com.gensym.shell.util` package provides managers and listeners that handle multiple connections to G2.

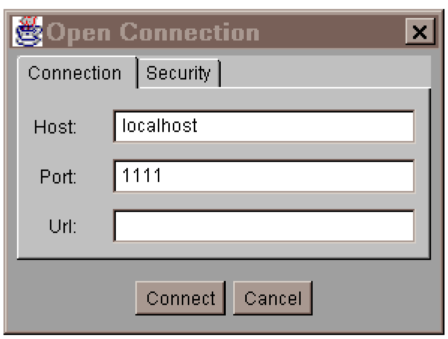
For detailed information, see “Creating and Managing Connections to G2” on page 236.

Shell Dialogs and UI Controls

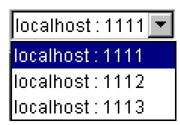
Telewindows2 Toolkit provides several dialogs and UI controls, which you can use directly in an application shell to allow users to perform these common interactions:

- Connecting and logging in to a secure or unsecure G2 through a tabbed dialog.
- Customizing tracing.
- Displaying and switching the current connection.
- Displaying and switching the current user mode.

This figure shows examples of these dialogs and UI controls:



LoginDialog



HostPortPanel

UserModePanel

The dialogs and UI controls are located in these packages:

```
com.gensym.shell.dialogs
com.gensym.shell.util
```

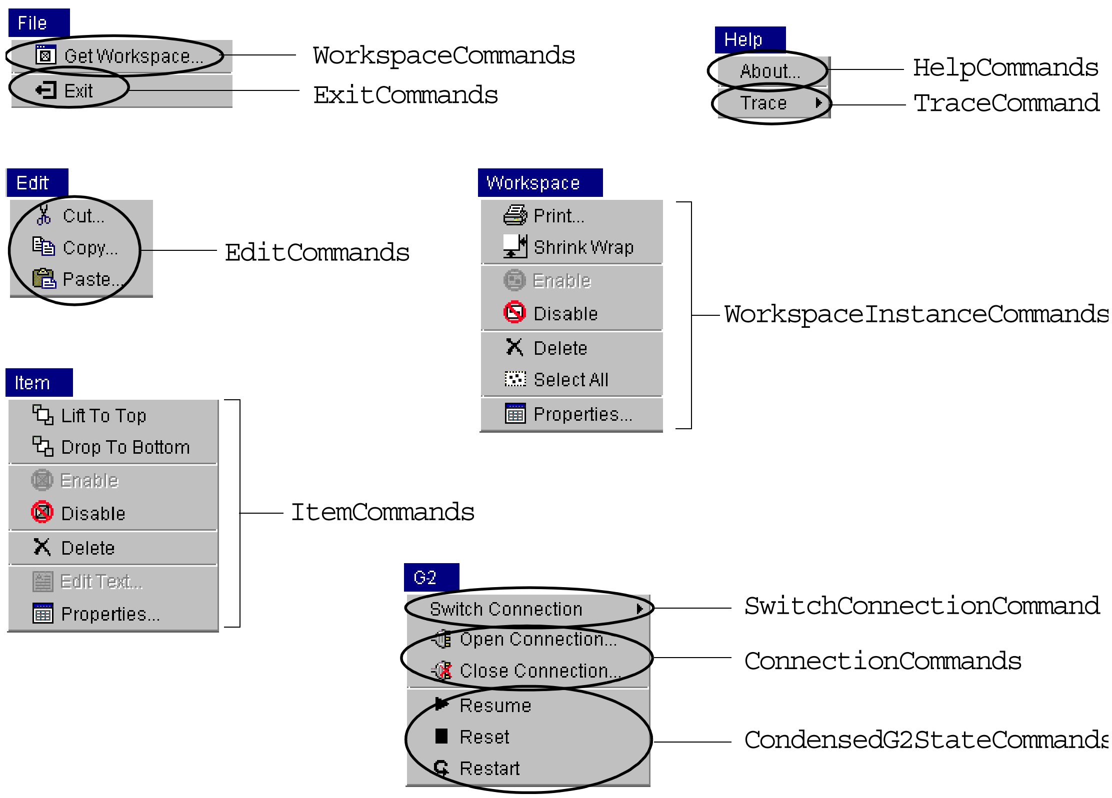
For detailed information, see Chapter 10, “Using Shell Dialogs and UI Controls” on page 259.

Shell Commands

TW2 Toolkit provides a number of commands, which you can add directly to your application menus and/or toolbars to support these common interactions with G2:

- Opening, closing, and switching between G2 connections.
- Getting named KB workspaces and creating new workspaces.
- Controlling the G2 run state.
- Performing standard cut/copy/paste operations on items on a KB workspace.
- Performing standard G2 interactions with items on a KB workspace.
- Interacting with KB workspaces.
- Getting help.
- Customizing tracing.
- Exiting the application.

The following figures show examples of the commands located in the `com.gensym.shell.commands` package as they appear in the TW2 Toolkit default application shell menus:



For detailed information about these commands, see Chapter 11, “Using Shell Commands” on page 271.

Telewindows2 Toolkit Default Application Shell

Telewindows2 Toolkit provides a default application shell, which is an example of a simple user interface for running G2 applications. The source code for the `Shell` class and its associated classes are located in the following directory:

NT: `%SEQUOIA_HOME%\classes\com\gensym\shell\`

UNIX: `$SEQUOIA_HOME/classes/com/gensym/shell/`

The TW2 Toolkit default application shell:

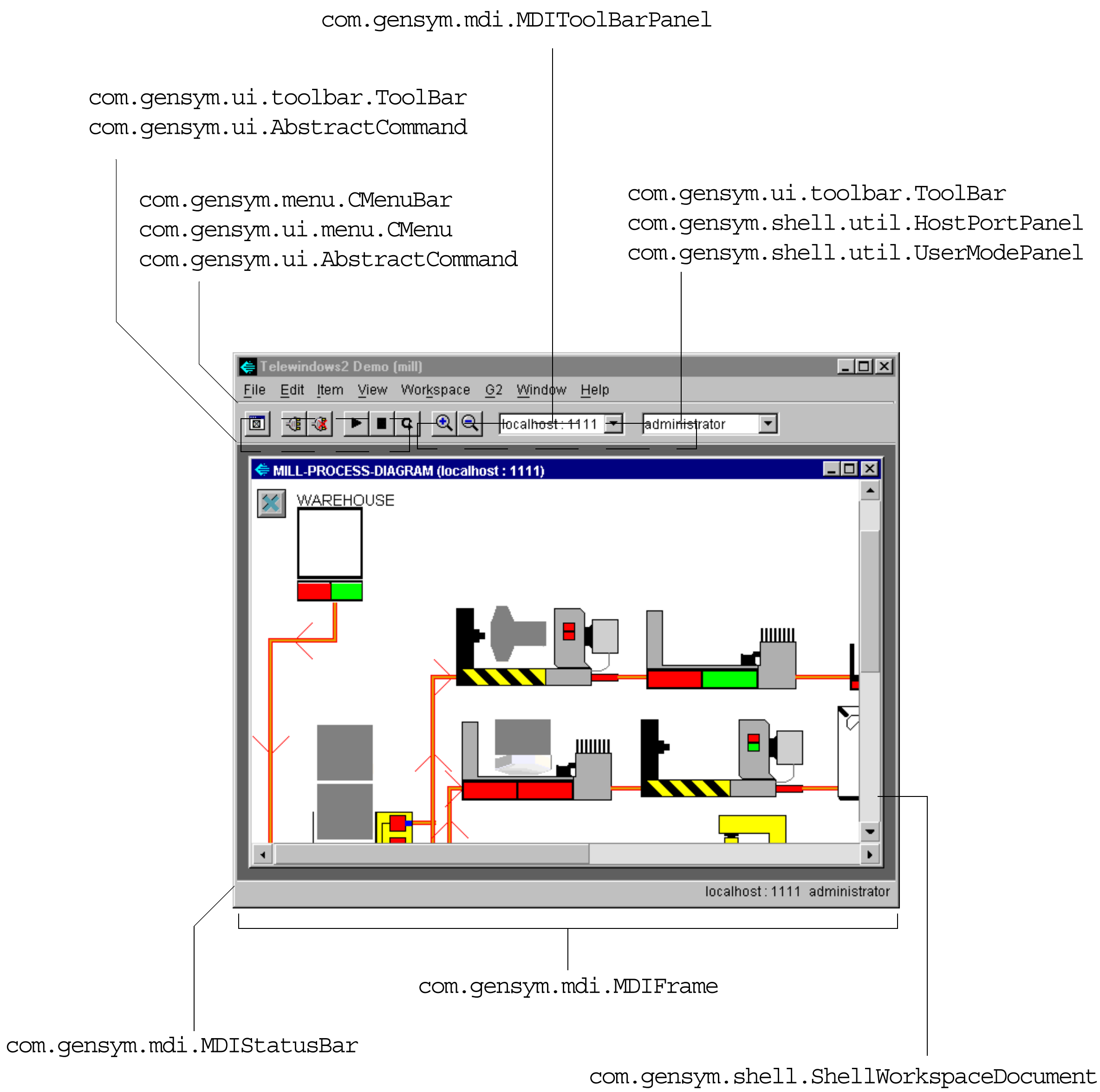
- Allows connecting to multiple G2s and switching between those connections.
- Simultaneously displays multiple workspace views, each in its own document window within an MDI application frame.
- Responds to programmatic show and hide KB workspace events in G2.
- Allows choosing named KB workspaces and navigating between those workspaces.
- Provides its own type of `MDIDocument` called `ShellWorkspaceDocument`, which:
  - Displays a workspace view.
  - Provides a context-specific menu bar that includes the Edit, Item, Workspace, and Window menus, in addition to the default menus.
  - Handles all aspects of managing the document when the connection to G2 closes or switches, and when the KB workspace in G2 is deleted.
- Provides its own type of `WorkspaceDocumentFactory` called `ShellWorkspaceDocumentFactoryImpl`, which generates a `ShellWorkspaceDocument`.

The source code used to create the TW2 Toolkit shell is available to you as an example of the kind of application you can build. The techniques that this shell uses are applicable for building any multiple connection, MDI application.

For a walk-through of the TW2 Toolkit default application shell end user interface, see Chapter 2, “Guided Tour of the Telewindows2 Toolkit Shell” on page 33.

For a walk-through of the source code for the shell, see Chapter 12, “Understanding the Telewindows2 Toolkit Shell” on page 301.

Here is the TW2 Toolkit default application shell that appears when you display a KB workspace.



Using Telewindows2 Toolkit Demonstrations for Java

Telewindows2 Toolkit includes numerous demonstrations illustrating various functionality for Java programmers. These demos show how to use Java Beans components, Java UI components, and Java application classes to build applets and applications that connect to a G2 server, display workspace views, and manipulate data.

These demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos`

To run the demos, you must either:

- Place the current version of G2 as the first G2 in your `PATH` environment variable.
- Define the `SEQUOIA_G2` environment variable to point to this version of G2.

A number of the demos make use of TW2 Toolkit components exclusively. These demos are described in the introduction to and throughout the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

Others demos are designed to illustrate how to use TW2 Toolkit application classes, although some of these demos also use TW2 Toolkit components.

The table below lists the source code location of each Java demo that uses TW2 Toolkit application classes and provides a description of each:

Java Demos

Source Code	Description

NT:
`standarddialogs\DlgTestApp.java`
UNIX:
`standarddialogs/DlgTestApp.java`
	Creates a Java frame that creates and launches informational dialogs and dialogs that accept user input.

NT:
`palettedemo\rundemo.bat`
UNIX:
`palettedemo/rundemo.sh`
	Shows how to create a palette of G2 objects and a native palette directly from a GFR palette.

Java Demos

Source Code	Description

NT:
`wksppanel\`
`SimpleWorkspaceApplication.java`
UNIX:
`wksppanel/`
`SimpleWorkspaceApplication.java`
	Creates a TW2 Toolkit UI application that lets you connect to a single G2 and display workspace views within a multiple workspace panel.

NT:
`wksppanel\BrowserApplication.java`
UNIX:
`wksppanel/BrowserApplication.java`
	Creates a TW2 Toolkit application that allows you to connect to a single G2 and display workspace views within a multiple workspace panel inside a single document frame.

NT:
`singlecxnsdiapp\`
`BrowserApplication.java`
UNIX:
`singlecxnsdiapp/`
`BrowserApplication.java`
	Creates a TW2 Toolkit application that allows you to connect to a single G2 and display workspace views within a single document frame.

NT:
`singlecxnmdiapp\`
`SingleConnectionApplication.java`
UNIX:
`singlecxnmdiapp/`
`SingleConnectionApplication.java`
	Creates a TW2 Toolkit application that allows you to connect to a single G2 and display workspace views within a multiple document frame.

NT:
`multiplecxnsdiapp\`
`WorkspaceBrowserApp.java`
UNIX:
`multiplecxnsdiapp/`
`WorkspaceBrowserApp.java`
	Creates a TW2 Toolkit application that allows you to connect to multiple G2s and display workspace views within a single document frame.

Java Demos

Source Code	Description
NT: `multiplecxnmdiapp\Shell.java` UNIX: `multiplecxnmdiapp/Shell.java`	Creates a TW2 Toolkit application that allows you to connect to multiple G2s and display workspace views within a multiple document frame.
NT: `classes\com\gensym\shell\Shell.java` UNIX: `classes/com/gensym/shell/Shell.java`	Shows the source code for Telewindows2 Toolkit default application shell.

Guided Tour of the Telewindows2 Toolkit Shell

Gives a guided tour of the end user features of the Telewindows2 Toolkit default application shell, which serves as an example of the type of client user interface you can build for G2 applications, using Telewindows2 Toolkit application classes.

Introduction

This chapter provides a guided tour of the **Telewindows2 Toolkit default application shell** for Java, a client user interface built in Java for running G2 applications. This application is also called the **TW2 Toolkit shell** or just the

shell. The shell allows you to connect to multiple G2 applications and to navigate between KB workspaces.

The chapter is structured as a tutorial; read it sequentially and follow the sets of steps in order. To go directly to the code used to implement this shell, see Chapter 12, "Understanding the Telewindows2 Toolkit Shell" on page 301.

This tutorial assumes you are running the G2 server on the same computer as the client.

Running the Telewindows2 Toolkit Shell

The TW2 Toolkit shell is based on this class:

```
com.gensym.shell.Shell
```

You can run the shell either:

- From the `shell` DOS batch file or UNIX shell script, depending on your platform, which is located in the `bin` directory of your TW2 Toolkit product directory.
- As a Java application, using the fully qualified class name.

When running the shell, you can provide command-line arguments for connecting to G2 and specifying the frame geometry.

Note Before you can run the shell, be sure you have installed all supporting software and set all required environment variables as described in the `readme.tw2.html` file.

Running the Shell as a Java Program

You will now run the TW2 Toolkit shell as a Java application, using command-line arguments.

The following example shows a partial list of the supported command-line arguments. For a complete list, see "Application Foundation Classes" on page 227.

To run the shell as a Java program:

➔ Enter the following command in a DOS command window or UNIX shell, depending on your platform:

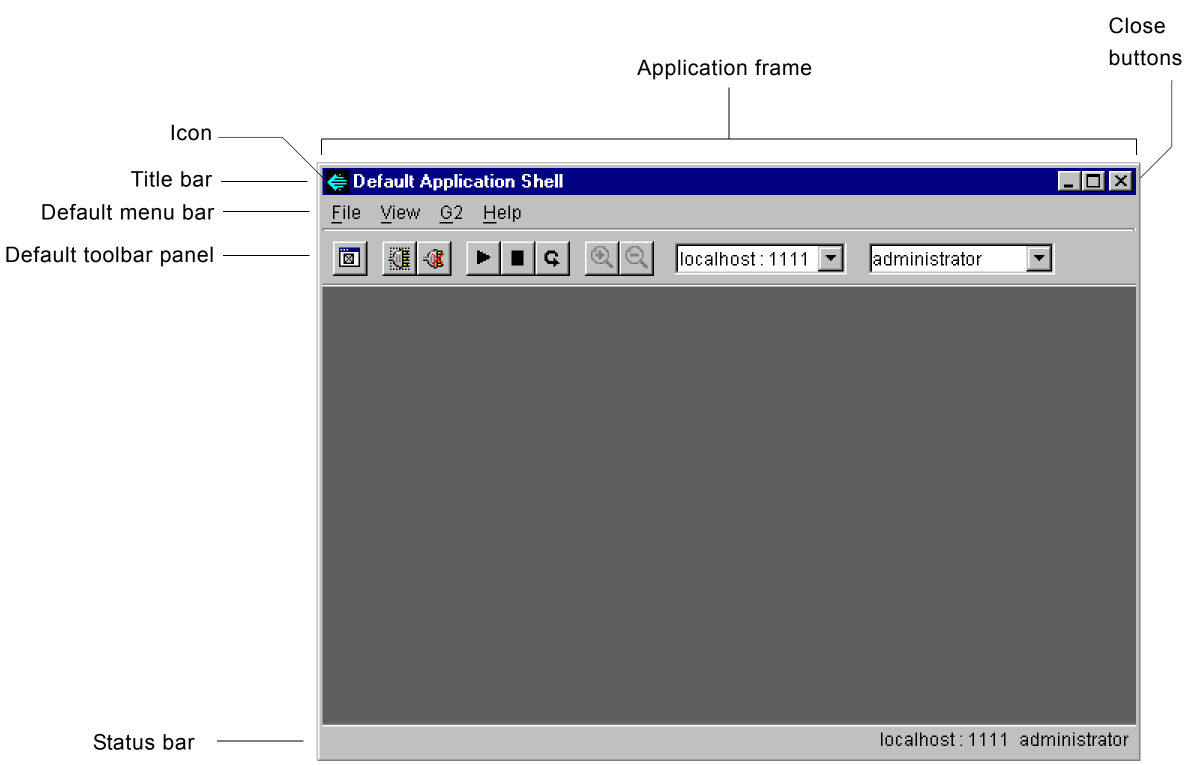
```
java com.gensym.shell.Shell
   [-title title -host host-name -port port-number
    -geometry widthXheight[+x+y][-x-y] ]
```

Command-Line Argument	Description
title	The title of the application's window as a string.
host-name	The name of the computer on which the G2 server is running.
port-number	The port on which the G2 server is running.
widthXheight [*+x+y*] [*-x-y*]	The width and height in pixels of the application window, separated by an "x", with optional x and y offsets from any corner of the screen.

For example, enter the following command to start the shell. This command connects the shell to the G2 server running on the local machine on port 1111, and places the window, which is 600x400 pixels, in the top-left corner of the screen, which is the default.

```
java com.gensym.shell.Shell -title "Default Application Shell"
   -host localhost -port 1111 -geometry 600x400+0+0
```

Here is the shell you will see:



Telewindows2 Toolkit Shell Features

The TW2 Toolkit shell has these features:

- An application frame with the Gensym logo as the icon in the upper-left corner of the frame for minimizing, maximizing, and closing the window.
- A default menu bar with these menus:
  - File menu for getting a KB workspace and exiting the application.
  - View menu for zooming a KB workspace.
  - G2 menu for opening, closing, and switching G2 connections, and for controlling the G2 run state.
  - Help menu with an About dialog and trace facilities.
- A default toolbar with:
  - Buttons for getting a KB workspace, connecting to and disconnecting from G2, controlling the G2 run state, and zooming in and out.
  - Choice boxes for switching the G2 connection and setting the user mode.
- A status bar.
- Mnemonics for all menu commands and shortcuts for certain menu commands.

Exiting the Shell

The TW2 Toolkit shell provides standard features for closing the application.

To close the shell:

➔ Click the close button in the upper-right corner of the window.

or

➔ Click the Gensym logo in the upper-left corner of the window and choose Close.

or

➔ Choose Exit from the File menu.

Running the Telewindows2 Toolkit Demo

You can use the TW2 Toolkit shell to view a demo G2 application by using the `t2demo` DOS batch file or UNIX shell script, depending on your platform. The batch file or shell script:

- Runs G2 on the local machine, using the default port, 1111.
- Loads `mill.kb`, located in the `kbs` directory of your TW2 Toolkit product directory.
- Runs the TW2 Toolkit shell and connects to the G2 application.

You can also run the demonstration manually by starting G2, loading the KB, running the shell, and connecting to G2.

Note Before you can run the demo, be sure that you have a valid `g2.ok` file that G2 can locate. For example, you can create a `G2V51_OK` environment variable that points to the location of your `g2.ok` file, you can specify the `-ok` command-line argument, or you can place the `g2.ok` file in your G2 directory.

First, you will run the demonstration manually, so you become familiar with this technique, then you will run it from the batch file or shell script, depending on your platform.

Running the Demo Manually

To run the demo manually, run the TW2 Toolkit shell, then start G2 and load the demo KB.

To run the TW2 Toolkit shell from a DOS batch file or UNIX shell script:

➔ Run the `shell` DOS batch file or UNIX shell script as follows, depending on your platform:

On Windows NT platforms:

➔ Double click the `shell.bat` batch file in the `bin` directory of your TW2 Toolkit product directory.

or

➔ From a DOS window, run the `shell` batch file from the `bin` directory of your TW2 Toolkit product directory.

On UNIX platforms:

➔ In a UNIX shell, run the `shell` script from the `bin` directory of the TW2 Toolkit product directory.

To run the G2 demo:

1 Run the `g2` executable as follows, depending on your platform:

➔ **On Windows NT platforms**:

Double-click the `g2.exe` executable file in your G2 product directory.

or

➔ From a DOS window, run the `g2` executable from your G2 product directory.

or

➔ Double-click the `start_g2.bat` file in the `bin` directory of your TW2 Toolkit product directory.

On UNIX platforms:

➔ In a UNIX shell, run the `g2` executable from your G2 product directory.

2 Load `mill.kb` from the `kbs` directory of your TW2 Toolkit product directory.

You should now see two application windows and two DOS command windows:

- A G2 application window with a schematic diagram of a mill application and its associated command window.
- The TW2 Toolkit shell with a top menu bar, toolbar, and its associated command window.

Connecting to G2 from the Client

Once you are running the shell and G2, you must connect manually to the G2 from the shell by using a menu choice or toolbar button.

When you connect to a secure G2 from the shell, you make a login request by providing a user name, user mode, and password.

By default, the shell logs you on in administrator mode, which allows you to access any G2 application.

To connect to G2 manually:

➔ Do one of the following:

Menu bar:

1. Choose Open Connection from the G2 menu.
2. Choose the Connection tab in the dialog.
3. Enter the Host and Port of the G2 to which you want to connect.

Toolbar:

1. Click the Open Connection button on the toolbar: 
2. Specify the host and port in the dialog.

The Mill application is running on the host computer named `localhost` on port `1111`. G2 starts running the Mill application and displays the top-level workspace.

When the shell is connected, the toolbar shows the host machine and port, as well as the user mode of the current connection.

You will now exit both applications and run the demo from a file.

To exit both applications:

1. To exit G2, choose Main Menu > Pause, then choose Main Menu > Miscellaney > Shutdown G2.
2. To exit the TW2 Toolkit shell, choose File > Exit.

Running the Demo from a File

Now you will run the demo from a DOS batch file or UNIX shell script, depending on your platform, which perform all of the functions for you, including connecting to G2, using command line arguments.

Note Depending on the processing speed of your computer, the login attempt might time out before G2 finishes loading the KB. If this happens, you will receive an error message. Clicking the OK button in the error dialog displays the Open Connection dialog for you to attempt the login again.

To run the TW2 Toolkit demo from a DOS batch file or UNIX shell script:

1. Run the `t2demo` DOS batch file or UNIX shell script as follows, depending on your platform:

   On Windows NT platforms:

   ➔ Double click the `t2demo.bat` batch file in the `bin` directory of your TW2 Toolkit product directory.

   or

   ➔ From a DOS window, run the `t2demo` batch file from the `bin` directory of your TW2 Toolkit product directory.

   On UNIX platforms:

   ➔ In a UNIX shell, run the `t2demo` shell script from the `bin` directory of your TW2 Toolkit product directory.

2. Minimize the DOS command windows or UNIX shells for the G2 and TW2 Toolkit applications.

Caution Do not close the DOS command windows or UNIX shells, or the TW2 Toolkit and G2 applications will close.

Displaying Workspace Views in the Client

When the TW2 Toolkit shell is connected to a G2 server, you can display and manipulate any named KB workspace in the G2 application from the client. The KB workspace appears in the shell as a **workspace view**, which is a client representation of a KB workspace.

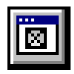
The workspace view, in turn, appears within a child frame, called a **workspace document**, of the overall application frame. The workspace document has its own context-sensitive menu bar, which the application automatically swaps in when the workspace document gains focus. The workspace document uses the default toolbars of the application frame.

You can view multiple copies of the same KB workspace by choosing the same named workspace multiple times, if desired.

Getting a Workspace View

To get a workspace view:

1 Do one of the following:

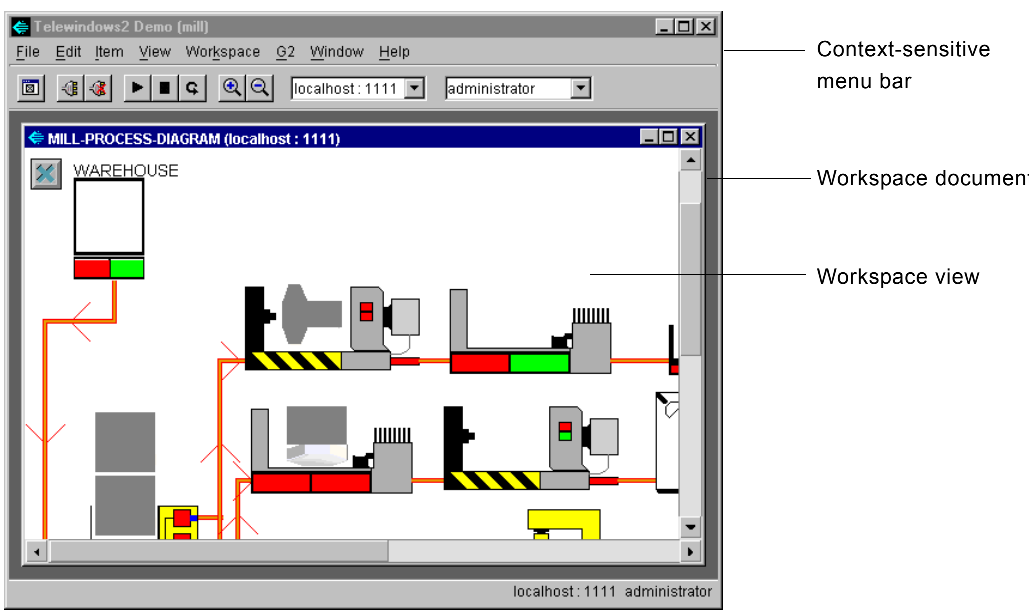
Menu bar:	Choose Get Workspace from the File menu.
Toolbar:	Click the Get Workspace button on the toolbar: 

2 Select mill-process-diagram from the list of named workspaces and click OK.

The shell creates and displays a child document in which it displays the selected workspace view. Notice that the workspace document has its own context-specific menu bar that includes these additional top-level menu choices:

- Edit
- Item
- Workspace
- Window

You will see a view of a schematic diagram of a milling application that creates bolts on an assembly line, which is running in G2:



When the shell displays the workspace view, G2 JavaLink automatically loads the necessary visual information about each G2 item into Java, thereby making it available for display. In addition, the shell can get and set attributes, and call methods on all the items in the workspace view, as needed. The workspace view obtains non-visual information, such as attribute values, only when the client requires that information.

You can get a handle on selected items in a workspace view and call methods on those items in any Java application, using JavaLink methods. If you need to get or set item properties, or call methods on a G2 item directly, without going through a workspace view, you can manually download Java class definitions for any G2 class to use in your Java application. You can also create Java Beans from G2 classes for use in any JavaBeans-compliant visual programming environment.

For more information, see these G2 JavaLink guides:

- *G2 DownloadInterfaces User's Guide*
- *G2 Bean Builder User's Guide*

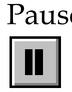
Controlling the G2 Run State from the Client

The TW2 Toolkit shell provides menu choices and toolbar buttons for controlling the G2 run state. You can pause, resume, restart, reset, and start G2 from the client through the menu or toolbar.

The Mill application is initially running.

To pause and resume G2 from the client:

1 Do one of the following to pause the KB:

Menu bar:	Choose Pause from the G2 menu.
Toolbar:	Click the Pause button on the toolbar: 

2 Do one of the following to resume the KB:

Menu bar:	Choose Resume from the G2 menu.
Toolbar:	Click the Resume button on the toolbar: 

G2 pauses and resumes running the KB, and notifies the shell that the run state has changed. The shell updates the buttons to reflect the current G2 run state.

Interacting with Items in Workspace Views

Workspace views in a client support most of the same features that KB workspaces in G2 support. You interact with items in a workspace view by using standard windowing techniques. For example, clicking an item in a workspace view selects the item, and clicking the right mouse button on an item displays its popup menu.

For information about the differences in behavior between workspace views and KB workspaces, see Chapter 10 “The Workspace View User Interface” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

Displaying the Popup Menu for an Item

You use an item’s popup menu to perform the same operations that you perform through the item menu in G2. These operations include:

- Cutting, copying, and pasting the item, using the clipboard.
- Editing the name of the item.
- Deleting the item.
- Enabling and disabling the item.
- Creating a subworkspace for the item.
- Rotating and reflecting the icon for the item.
- Editing the color of the icon regions for the item.
- Lifting the item to the top and dropping it to the bottom.
- Describing the item.
- Editing the attribute display of an item.
- Displaying the item’s Properties dialog.

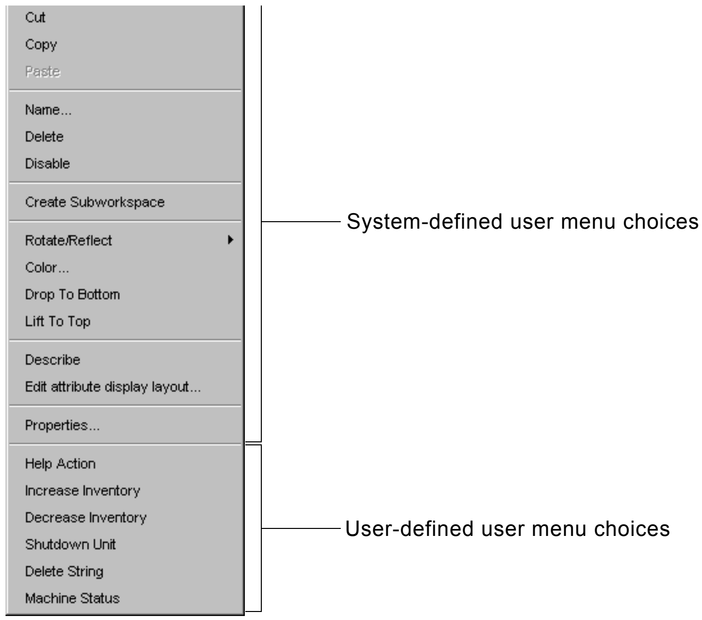
The workspace view automatically creates menu choices in the popup menu for all relevant system-defined menu choices and all user-defined menu choices for the G2 class. Clicking a user menu choice on the item in the client executes the action in the G2 server, which updates the representation of that item in the TW2 Toolkit client appropriately.

Note User menu choices only appear in the popup menu if G2 is running.

You will now display the popup menu on the Warehouse item.

To display the popup menu for an item:

➔ Click the right mouse button on the Warehouse in the upper-left corner of the workspace view to display its popup menu:



Editing Item Properties

The TW2 Toolkit shell automatically generates a **properties dialog** for each item in the workspace view when the user chooses Properties from the item's popup menu. The properties dialog is analogous to the G2 attributes table for an item.

The automatically generated properties dialog uses a variety of controls, depending on the type specification of the attributes of the G2 class definition. When you edit an attribute that requires G2 syntax, the properties dialog launches a native, **syntax-guided text editor**.

For information on how to use the text editor, see Chapter 11 "Using the Text Editor" in the*Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

Editing the properties of an item through the dialog edits the corresponding attribute in the G2 server, which updates the representation of the item in the client appropriately.

You will now edit the Names property of the Warehouse in the mill-process-diagram workspace view through the properties dialog.

To edit the attributes of an item in a workspace view:

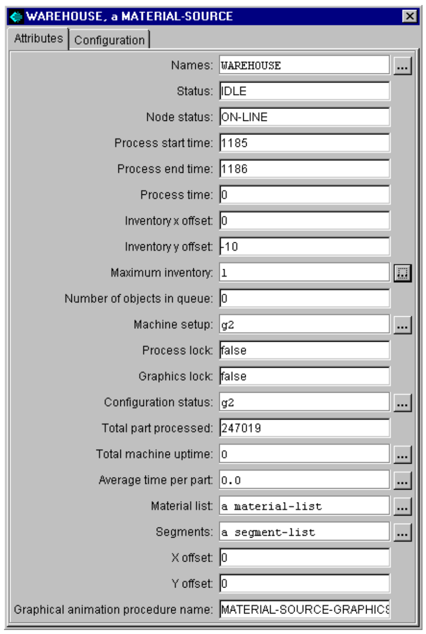
1 Do one of the following to display the automatically generated properties dialog for the item:

➔ Choose Properties from the Warehouse's popup menu.

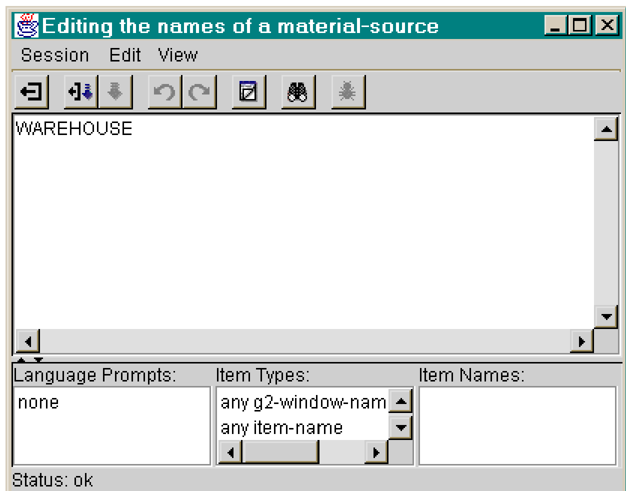
or

➔ Double-click the item.

You will see this dialog:



2 Click the button next to the Names edit box to display the syntax-guided text editor:



Note The button with ellipses appears next to all attributes that you cannot edit in place, which includes attributes with a grammar, attributes that contain subobjects, and color attributes. Editing an attribute that contains a subobject displays a Properties dialog for the subobject.

---

3 Edit the name to be `new-warehouse`.

4 Do one of the following to save the changes and exit:

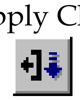
Menu bar:

➔ Choose Save, then choose Exit from the Session menu.

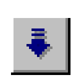
or

➔ Choose Exit from the Session menu, then click OK in the confirmation dialog that appears.

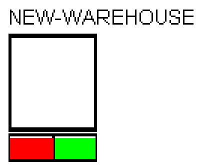
Toolbar:

➔ Click the Apply Changes and Exit button on the toolbar. 

or

➔ Click the Save button  followed by the Exit button. 

The attribute display of the name is updated in the G2 server. The representation of the item in the TW2 Toolkit client then updates to reflect the new name:



Item Configurations and User Modes

The workspace view reflects all the G2 item configurations that are active in the connected G2 at the time of the connection. For example:

- If the visible attributes in the G2 table have been restricted in the server, the automatically generated properties dialog hides those attributes in the client.
- If the user menu choices for a G2 class have been restricted, the popup menu for the item hides those menu choices in the client.
- If you configure the behavior of an item when you select it in G2, the item has the same behavior in the client.

You switch the user mode by entering an existing user mode in the choice box on the toolbar panel. Once you have entered an existing user mode, you can switch the user mode by choosing from the list of available modes.

To edit the user mode:

➔ Enter Developer in the user mode choice box.

The user mode changes in G2, which updates all aspects of the application that depend on the user mode through its item configurations. The client is notified of the change in user mode, which updates the user mode choice box in the client.

The user mode you entered now appears in the list of available modes in the choice box:

Custom Dialogs

You can replace automatically generated properties dialogs with **custom dialogs** for individual classes or instances. You create custom item properties dialogs by using the TW2 Toolkit dialog components in any Java programming environment. Because the TW2 Toolkit components that you use to create dialogs are JavaBeans compliant, you can use an JavaBeans-compliant visual programming tools, such as Symantec Visual Café or Borland J Builder.

You can also edit the way in which TW2 Toolkit automatically generates dialogs for items.

You can also create, launch, and manage dialogs for any purpose in your application, such as launching a dialog based on an event in the server or in the client.

For information on...	See...
Creating custom dialogs	Chapter 15 "Using Dialog Components" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.
Launching custom item properties dialogs	Chapter 16 "Launching Custom Item Properties Dialogs" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.
Customizing automatically generated dialogs	Chapter 17 "Customizing Automatically Generated Dialogs" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.
Launching and managing custom item properties dialogs	Chapter 18 "Launching General Dialogs" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.
Creating, launching, and managing Java dialogs	Chapter 4, "Using Standard Dialogs" on page 71.

Interacting with an Item from its Popup Menu

As described earlier, you can perform standard G2 operations on an item through its popup menu. In general, the system menu choices behave just as they do in G2, with the following exceptions:

- Cut, Copy, and Paste allow you to clone and transfer items between workspaces in the same G2; they do not work for cloning and transferring items between workspaces in different G2s. These commands replace the clone and transfer system menu choices in G2.
- Editing the item name displays the native text editor.

- Editing the item color displays a dialog that lets you choose from a palette a G2 color for each icon region.
- Describing the item displays a dialog in the client.

Experiment with the popup menu choices for the Warehouse item now.

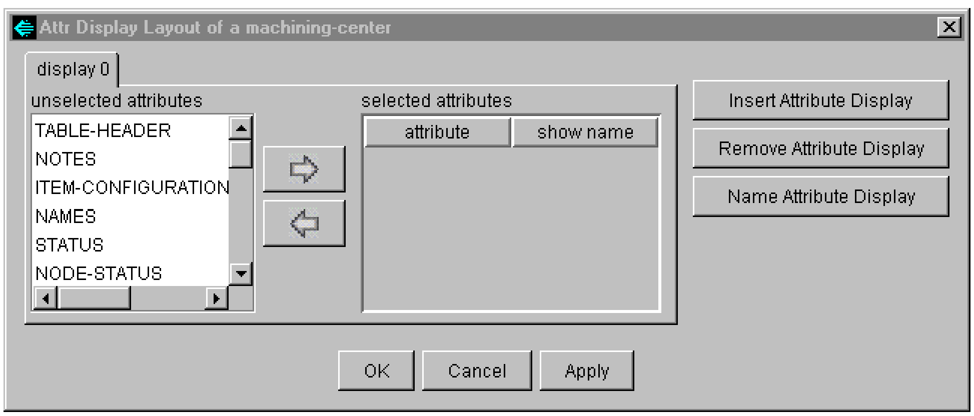
Editing Attribute Displays and Layout

You can edit the value of an attribute of an item by double-clicking the attribute display, which launches the text editor.

You can also edit which attribute displays appear next to an item and whether the attribute display includes the attribute name.

To edit the attribute display layout for an item:

1 Choose Edit Attribute Display Layout from the item popup menu to display this dialog:



Each attribute display can have one or more attributes and is displayed in its own tab page in the dialog.

2 Select an attribute from the unselected attributes list on the left, whose value you want to display next to the item, then click the right arrow button to move it to the selected attributes list.

3 Click the Show Name toggle button next to the selected attribute to display the attribute name with its value.

Tip You can resize the width of the Attribute and Show Name columns in the selected attributes list by dragging the border between the header of the two columns.

4 To add and remove attribute displays and corresponding tab pages, click the Insert Attribute Display and Remove Attribute Display buttons.

If you wish to rename the tab associated with an attribute display, click the Name Attribute Display button and enter a name.

5 Click the OK button to add the attribute display.

Selecting, Moving, and Resizing Items

You use standard windowing techniques for selecting, deselecting, moving, and resizing items in a workspace view.

For additional information on working with items in a workspace view, see Chapter 10 “The Workspace View User Interface” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

To select an item:

➔ Click the item.

or

➔ Drag the mouse over a region of the workspace view to select all the items in the region.

To add items to a selection:

1 Select one or more items, using the standard techniques.

2 Hold down the Shift key while selecting additional items.

To deselect an item:

➔ Shift-click the selected item.

To move an item:

➔ Select the item, then drag it to a new location.

To resize an item:

➔ Select the item, then drag the selection handles.

To select all the items in a workspace view:

➔ Choose Select All from the Workspace menu.

Interacting with Workspace Views

You can edit the properties of and interact with a KB workspace from the popup menu for the workspace view or from the top-level Workspace menu. The popup menu for a workspace view is similar to that of an item, with these additional menu choices:

- New Item — Displays a palette of items from which you can create items on the KB workspace, and allows you to edit the classes of items on the palette.
- Clone — Clones the KB workspace.
- Shrink Wrap — Shrink wraps the KB workspace.
- Print — Displays a standard print dialog for printing the workspace view to a printer.

In addition, you can scale the workspace view in the client.

In general, all interactions with the workspace view in the client modify the KB workspace in the G2 server. The exceptions are printing and scaling the workspace view, which affect only the client.

For additional information on interacting with workspace views, see Chapter 10 "The Workspace View User Interface" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

Editing KB Workspace Properties

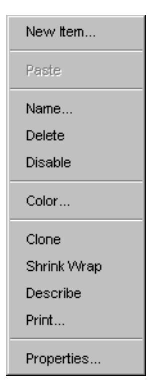
To edit the properties of a KB workspace:

1 Do one of the following:

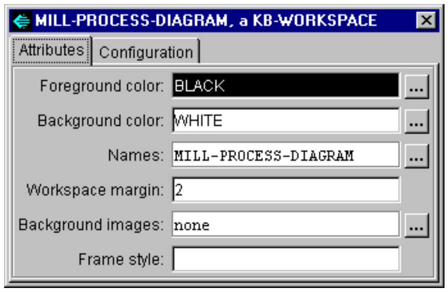
➔ Choose Properties from the Workspace menu.

or

a Click the right mouse button anywhere on the background of the workspace view to display its popup menu:



b Choose Properties to display the automatically generated properties dialog for the KB workspace:



2 Edit the desired attribute of the KB workspace.

3 Close the properties dialog.

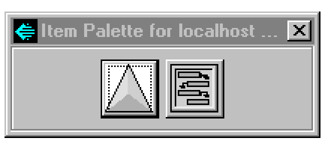
The workspace name is updated in the G2 server, and the new name is reflected in the TW2 Toolkit client.

Creating New Items on a KB Workspace

You can create new items on a KB workspace by using a palette. You can edit the palette to include those items you create most often.

To create a new item:

1 Choose New Item from the popup menu for a workspace view to display this default palette:



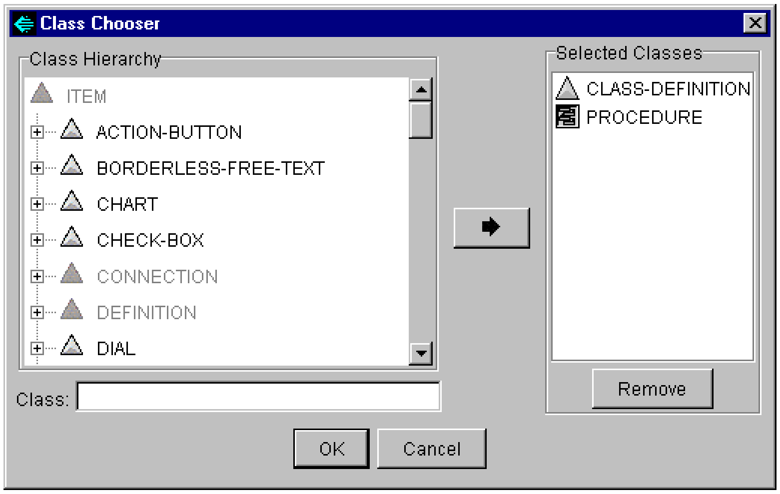
2 Click an item on the palette, then click the cross-hairs that appear anywhere in the workspace view to place the item.

To edit the palette of objects:

1 Choose New Item from the popup menu for a workspace view.

2 Click the right mouse button on the background of the Item Palette and choose Edit Classes.

The following Class Chooser dialog appears, which contains a tree view of the class hierarchy of all system-defined and user-defined classes, and a list of selected classes to display in the palette:



3 Click the plus sign next to a class to expand the tree to show subclasses.

If a class does not have any subclasses, clicking the plus sign simply removes the plus sign.

Tip If you know the name of the class you wish to include on the palette, enter it directly in the Class field.

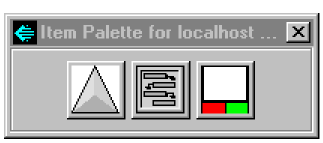
---

4 To include a class in the Item Palette, select the item and click the right arrow button to move it to the Selected Classes list.

5 To remove a class from the palette, select the item in the Selected Classes list and click the Remove button.

6 Click OK.

Here is an Item Palette that includes the user-defined material-source class:



Cloning a KB Workspace

Cloning a workspace view in the client creates a duplicate KB workspace in G2 and shows that workspace in the client.

To clone a KB workspace:

➔ Choose Clone from the popup menu for the workspace view.

Shrink Wrapping a KB Workspace

When you shrink wrap a workspace view in the client, the borders of the KB workspace in G2 shrink to just contain the items. Shrink wrapping a workspace view in the client has the same effect, except it does not shrink wrap the workspace document that contains the workspace view.

To adjust the workspace document to fit the view, drag the corner until it just fits the workspace view.

To shrink wrap a KB workspace:

➔ Choose Shrink Wrap from the Workspace menu.

or

➔ Choose Shrink Wrap from the workspace view's popup menu.

Scaling a Workspace View

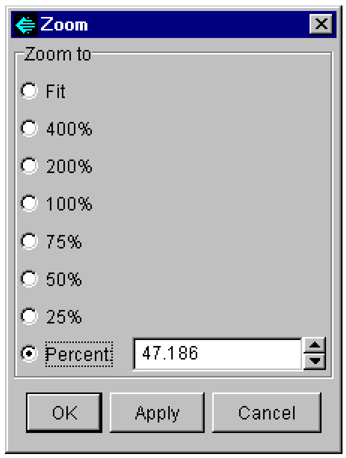
You can explicitly scale a workspace view to fit the dimensions of the workspace document, scale the view in standard increments, scale the view to a given scale that you enter, or incrementally scale the view in and out.

If you choose to scale the workspace view to fit, then dragging the corner of the workspace document scales the workspace view to fit the document, while maintaining the aspect ratio of the workspace view.

Scaling the workspace view in the client has no effect on the KB workspace in the G2 server.

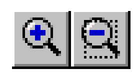
To scale a workspace view:

➔ Choose Zoom from the View menu to display the following dialog, then choose Fit, choose a zoom scale, or enter a specific percent value, then click OK:

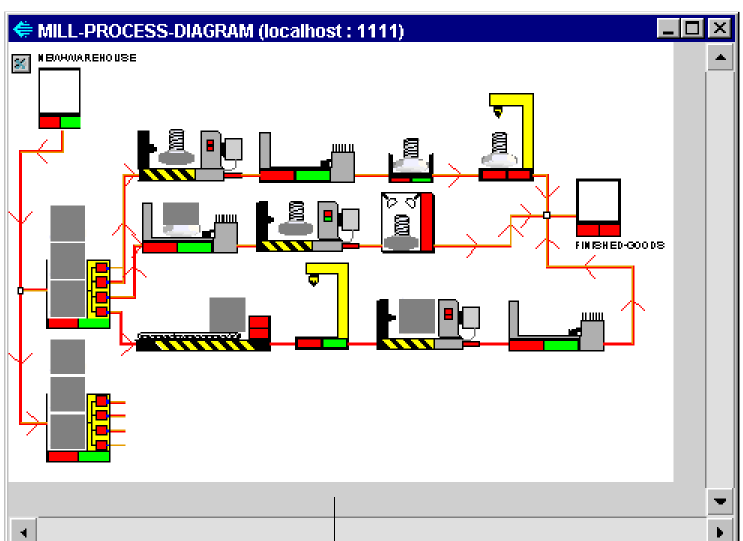


or

➔ Click the Zoom In or Zoom Out button on the toolbar.



Here is a workspace view that has been scaled:



Grey border is not part of the workspace view.

Notice that the workspace document that contains the view does not scale with the workspace view; sometimes, a grey border appears around the workspace view, which is not part of the workspace view.

Printing a KB Workspace

To print the KB workspace:

1 Do one of the following:

➔ Choose Print from the Workspace menu to display a standard print dialog for configuring the printer.

or

➔ Choose Print from the workspace view's popup menu.

2 Configure the Print dialog and click OK.

Connecting to Multiple G2 Applications from the Client

You can connect to one or more G2 applications and switch between them in the client. To do this, you must have sufficient Telewindows2 Toolkit licenses to connect to multiple G2s.

For example, you will now load on your local host the application named `sq-demos.kb`, located in the `kbs` directory of your Telewindows2 Toolkit product directory.

To connect to a second G2 application:

1 Launch a second G2 application on your local host from the command line and load `sq-demos.kb`, as follows:

NT:

```
g2 -kb %SEQUOIA_HOME%\kbs\sq-demos.kb
 -host localhost -tcpport 1234
```

UNIX:

```
g2 -kb %SEQUOIA_HOME%/kbs/sq-demos.kb
 -host localhost -tcpport 1234
```

2 Open a connection to the local host on port 1234 by using the Open Connection command on the G2 menu or the equivalent toolbar button.

Tip To determine the host and port of your G2 application, choose Main Menu > Miscellany > Network Info in your G2 application. If you do not specify a port when you launch G2 on your local host, G2 automatically assigns sequential port numbers, beginning with 1111.

Displaying Multiple Workspace Views for Different G2 Connections

When multiple G2 connections exist in the client, you can display multiple workspace views associated with different G2 servers simultaneously and switch between those views. Each time you switch workspace views, the shell automatically switches the current connection. You can also switch the connection manually.

The shell provides several ways of switching between multiple workspace views:

- Clicking anywhere in the workspace document to make it active.
- Choosing a named KB workspace from the Windows menu.
- Switching the connection manually and choosing Get Workspace.

You manage the window that contains a workspace view by using standard buttons to minimize, maximize, and close the window.

You can arrange the multiple windows vertically, horizontally, or in a cascade, using the Windows menu.

To display and arrange workspace views for different G2 connections:

1. Open a G2 connection to the host running `sq-demos.kb`.

   Notice that the workspace view of the mill-process-diagram workspace no longer has focus.

2. Display the solar-system workspace.

   You will see a schematic diagram of a solar system.

3. Click the title bar of the mill-process-diagram workspace view to bring it to the foreground.

   Notice that the current connection has changed, as the toolbar indicates.

4. Choose solar-system from the list of available workspace views in the Window menu.

   The solar-system workspace is in the foreground again, and the connection has changed.

5 Choose Tile Vertically from the Window menu to display the two workspace views side-by-side.

6 Close both workspace views.

To switch the connection manually:

1 Do one of the following:

Menu bar:	Choose Switch Connection from the G2 menu and choose the desired connection from the cascading submenu.
Toolbar:	Choose the desired connection from the choice box on the toolbar: 

2 Choose Get Workspace from the menu or toolbar.

Using Menu Command Mnemonics and Shortcuts

The default shell supports mnemonics for all menu choices, which lets you execute the menu choice by entering Alt + <menu key> + <choice key>.

The conventions for the mnemonics follow the conventions of the native user interface for your platform. To determine the mnemonic for a menu choice, look for the underlined letter in the top-level menu and menu choice labels. For example, to execute the File > Get Workspace command, enter Alt + f + g.

In addition, the default shell supports keyboard shortcuts for the following menu choices:

Menu Command	Shortcut
Edit > Cut	Ctrl + x
Edit > Copy	Ctrl + c
Edit > Paste	Ctrl + v
Workspace > Print	Ctrl + p
Workspace > Select All	Ctrl + a

Exiting the Telewindows2 Toolkit Demo

You have finished the tutorial.

Exit the demonstration:

➔ Exit the shell and G2 to finish this tutorial.

3

Road Maps to Using This Guide

Gives a road map for where to go in this guide for information about building various types of applications, using Telewindows2 Toolkit application classes.

Introduction

This chapter provides several road maps to guide you through the chapters and sections in this manual.

The first road map describes the following two high-level tasks, while the subsequent road maps describe the specific tasks listed below each high-level task:

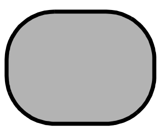
- Using Telewindows2 (TW2) Toolkit UI controls and containers, including:
  - Standard dialogs.
  - Menus, toolbars, and commands.
  - Palettes.
  - Multiple document interface (MDI) containers.
  - MDI document types that display views of G2 server data.

- Using TW2 Toolkit application foundation and shell classes to create:
  - Single document interface (SDI) and multiple document interface (MDI) applications that manage frames and connections as part of the API.
  - Dialogs and UI components that provide user interfaces for common interactions, such as logging on to the G2 server.
  - Commands that perform common actions, such as connecting to and disconnecting from the G2 server, and getting a KB workspace.

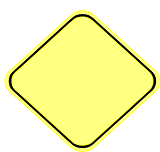
The page numbers in the maps provide references to the relevant chapters and sections in this guide.

Road Maps

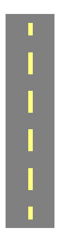
The road maps that follow use these symbols:



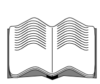
Shaded areas indicate categories of topics from which to choose. To view a detailed map of tasks for the topic, go to the page number of the category that most closely matches your application's needs.



Road signs indicate the task you want to perform.



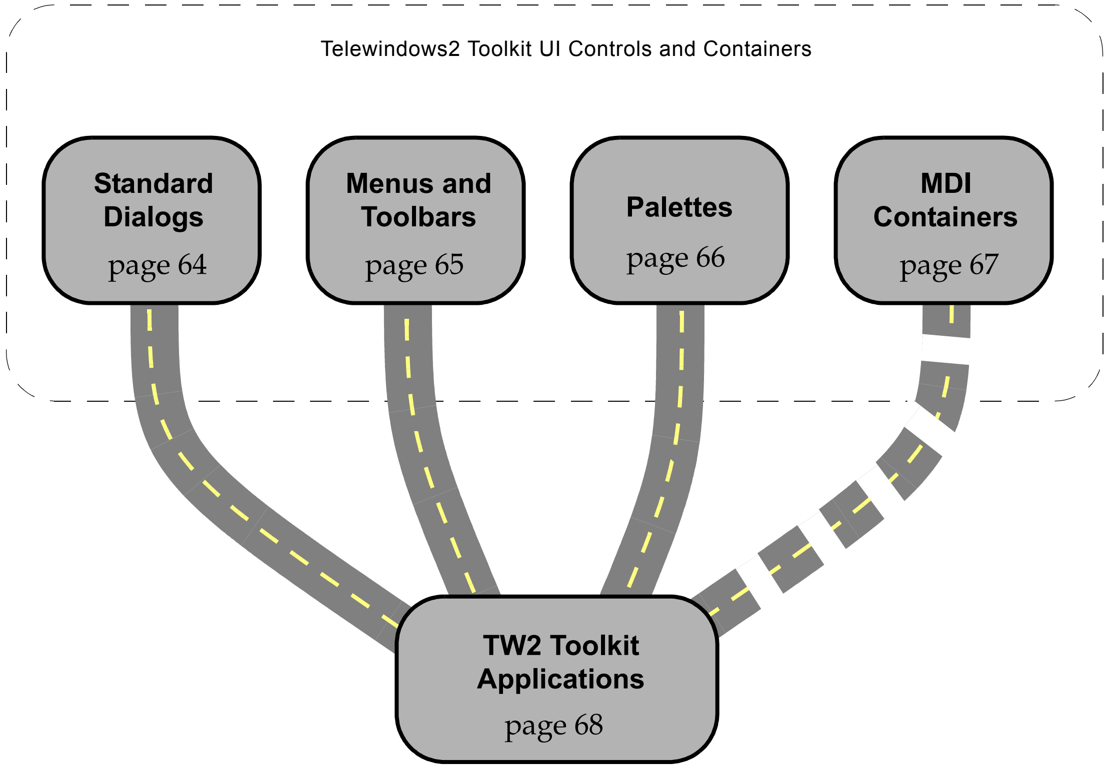
Roads indicate the dependency order of tasks and a relative increase in the amount of programming required to accomplish the task as you move from task to task. A broken road indicates a partial dependency. Start at the top/left of the map with the most fundamental task, then proceed along the roads to more advanced tasks, which depend on previous



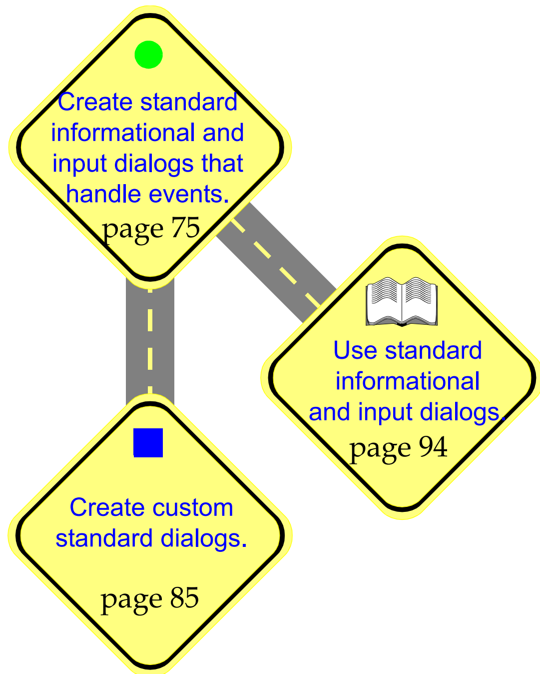
Book symbols indicate that the documentation provides reference information for the topic.

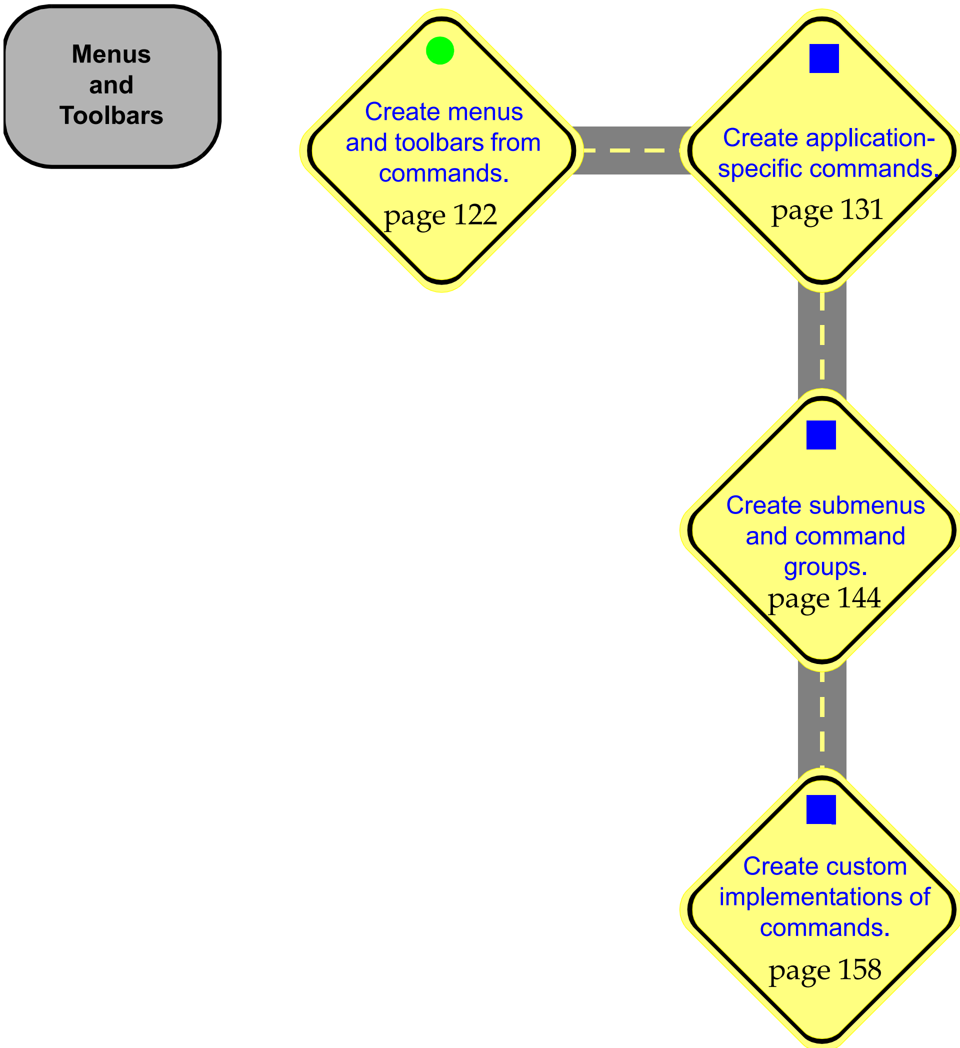
Circles indicate that the task involves making built-in TW2 Toolkit functionality visible to the user, which requires very little Java programming, typically, 1 - 5 lines of code. For example, creating a menu bar requires one line of code for each menu item.

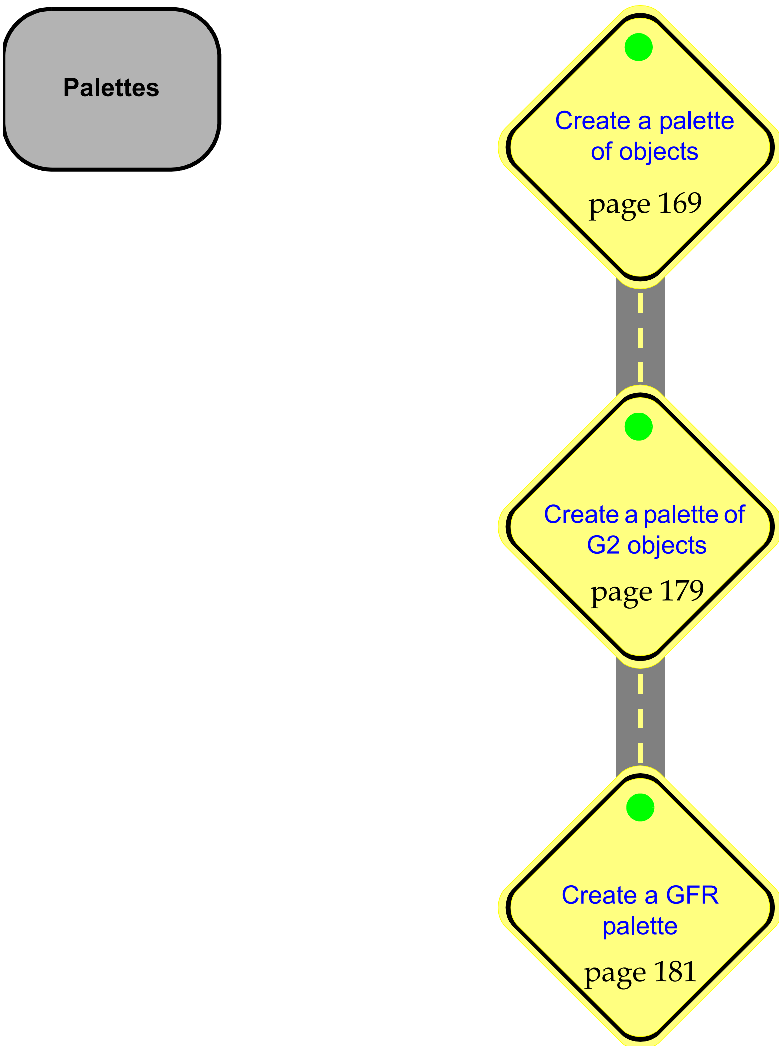
Squares indicate that the task involves implementing your own functionality, which requires somewhat more Java programming.

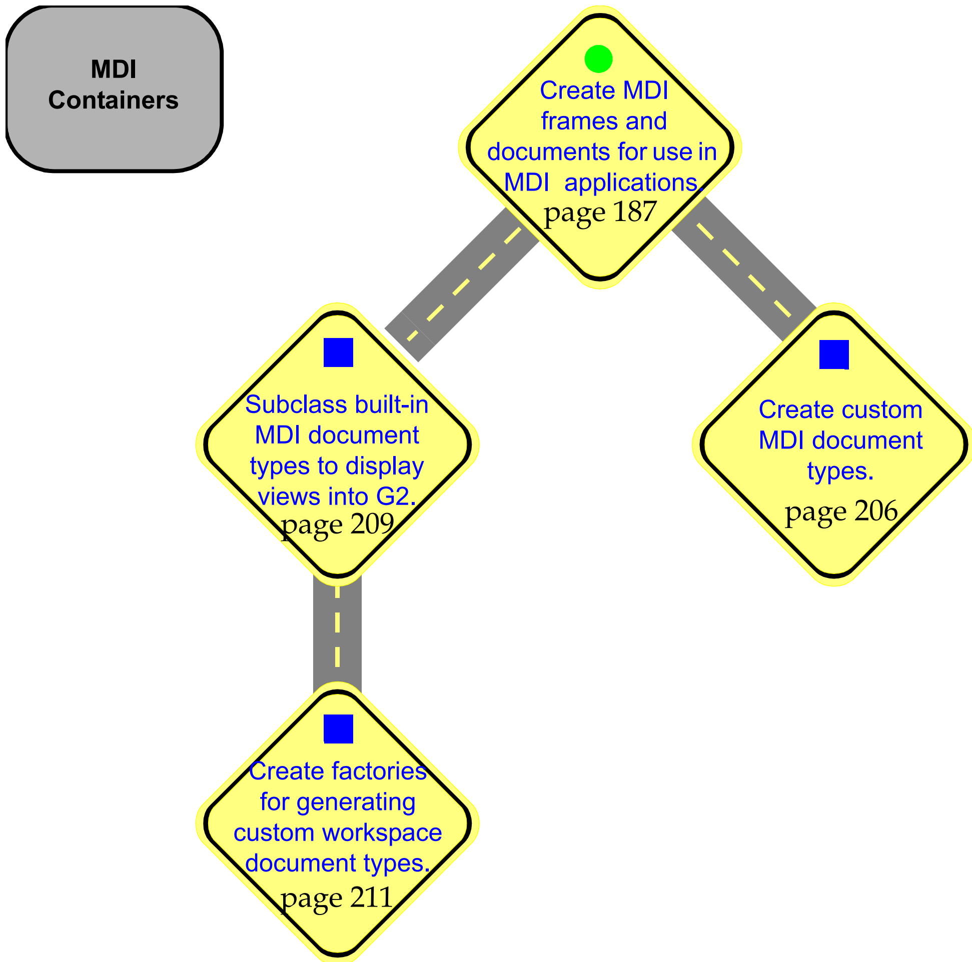

Telewindows2 Toolkit UI Controls and Containers
Standard Dialogs
page 64
Menus and Toolbars
page 65
Palettes
page 66
MDI Containers
page 67
TW2 Toolkit Applications
page 68

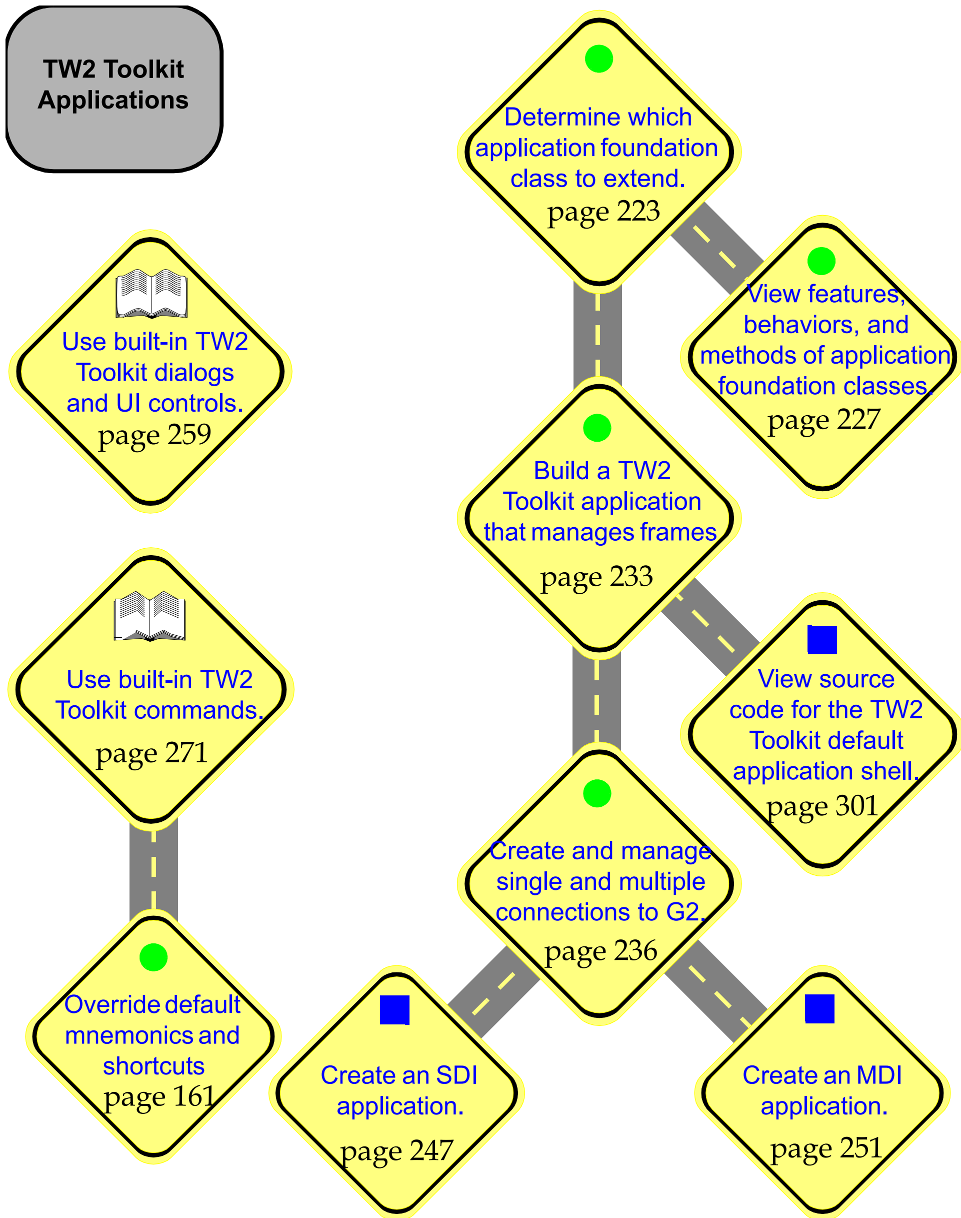
Standard Dialogs




Menus and Toolbars
Create menus and toolbars from commands.
page 122
Create application-specific commands.
page 131
Create submenus and command groups.
page 144
Create custom implementations of commands.
page 158


Palettes
Create a palette of objects
page 169
Create a palette of G2 objects
page 179
Create a GFR palette
page 181


MDI Containers
Create MDI frames and documents for use in MDI applications
page 187
Subclass built-in MDI document types to display views into G2.
page 209
Create custom MDI document types.
page 206
Create factories for generating custom workspace document types.
page 211


TW2 Toolkit Applications
Determine which application foundation class to extend.
page 223
View features, behaviors, and methods of application foundation classes.
page 227
Use built-in TW2 Toolkit dialogs and UI controls.
page 259
Build a TW2 Toolkit application that manages frames.
page 233
Use built-in TW2 Toolkit commands.
page 271
View source code for the TW2 Toolkit default application shell.
page 301
Create and manage single and multiple connections to G2.
page 236
Override default mnemonics and shortcuts
page 161
Create an SDI application.
page 247
Create an MDI application.
page 251

UI Controls and Containers

4

Using Standard Dialogs

Describes how to use standard information dialogs and dialogs that accept user input, and provides a reference for each dialog.

Introduction

The `com.gensym.dlg` package, which is part of G2 JavaLink, includes classes that you can use to create:

- **Informational dialogs** — Dialogs that display information to the user, which have a read-only text area, an OK button for dismissing the dialog, and, in most cases, an icon.
- **Input dialogs** — Dialogs that accept input from the user, which have dialog controls for specifying values and, in most cases, OK and Cancel buttons.

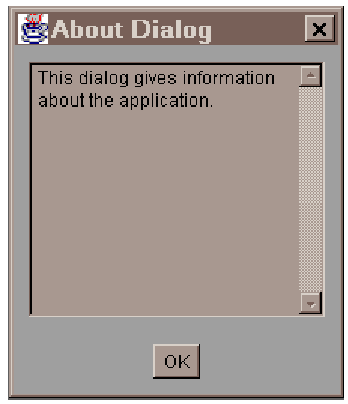
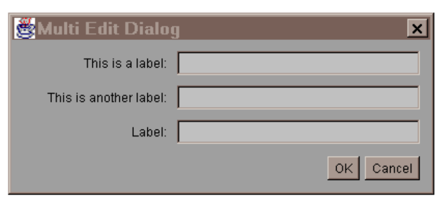
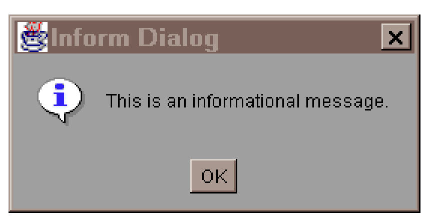
These classes are called **standard dialogs**, because they inherit their definition from this abstract class:

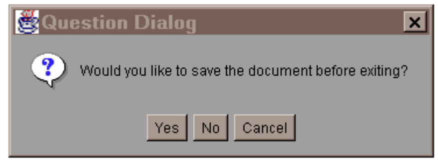
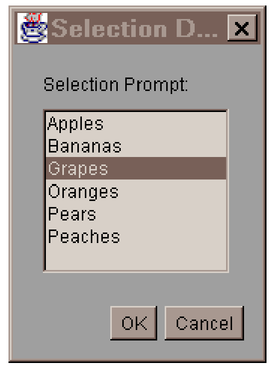
```
com.gensym.dlg.StandardDialog
```

You may use the classes in this package to create informational dialogs in your application, or you may use a Java dialog class such as `java.awt.Dialog` or `javax.swing.JDialog`.

Summary of Standard Dialog Classes

This table describes and gives examples of each standard dialog class:

<table>
<tr><th>Class</th><th>Description</th><th>Example</th></tr>
<tr><td><code>AboutDialog</code></td><td>Displays information about an application in a scrollable text area, typically invoked from a Help menu choice.</td><td>
</td></tr>
<tr><td><code>ErrorDialog</code></td><td>Displays error text to the user.</td><td>
</td></tr>
<tr><td><code>InputDialog</code></td><td>Obtains information from the user through one or more text fields.</td><td>
</td></tr>
<tr><td><code>MessageDialog</code></td><td>Displays message text to a user.</td><td>
</td></tr>
</table>

Class	Description	Example
`QuestionDialog`	Requests that the user respond to a question by clicking the Yes or No button.	
`SelectionDialog`	Displays a list of items from which the user can choose one or more items, depending on how the dialog is created.	
`WarningDialog`	Displays warning text to the user.	

For information on...	See...
Using the common features of standard dialog classes	“Using Standard Dialogs” on page 75.
Using individual standard dialog classes	“Standard Dialogs Reference” on page 94.

Standard Dialog Clients

Typically, you use standard dialogs in conjunction with a `StandardDialogClient`, which handles event notification when the user has clicked a button to dismiss the dialog. If a standard dialog provides more than one way to dismiss the dialog, you can determine which button the user has clicked to specify unique behavior for each button.

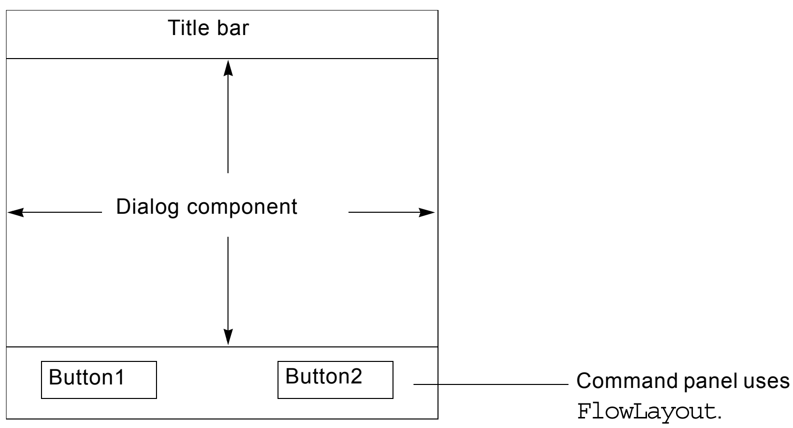
For details, see “Listening for Dialog Events” on page 77.

Dialog Layout

All standard dialogs consists of:

- A title bar.
- A single dialog component with one or more dialog controls, which are centered vertically.
- A command panel with one or more action buttons, which use a `java.awt.FlowLayout` as its layout manager.

The following diagram shows the layout of a standard dialog:



Custom Dialogs

You can customize the buttons, icon, and behavior of any standard dialog by extending one of the standard dialog classes. You can also customize the Java components that appear in the dialog by extending the `StandardDialog` class.

For details, see “Customizing Dialogs” on page 85.

Packages Covered

com.gensym.dlg

Interfaces

```
CommandConstants
StandardDialogClient
```

Classes

```
AboutDialog
ErrorDialog
InputDialog
MessageDialog
QuestionDialog
SelectionDialog
WarningDialog
```

Relevant Demos

The demo in the following directory, depending on your platform, uses the standard dialog classes:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\standarddialogs\DlgTestApp.java`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/standarddialogs/DlgTestApp.java`

Other demos in the `com.gensym.demos` package also uses standard dialog classes.

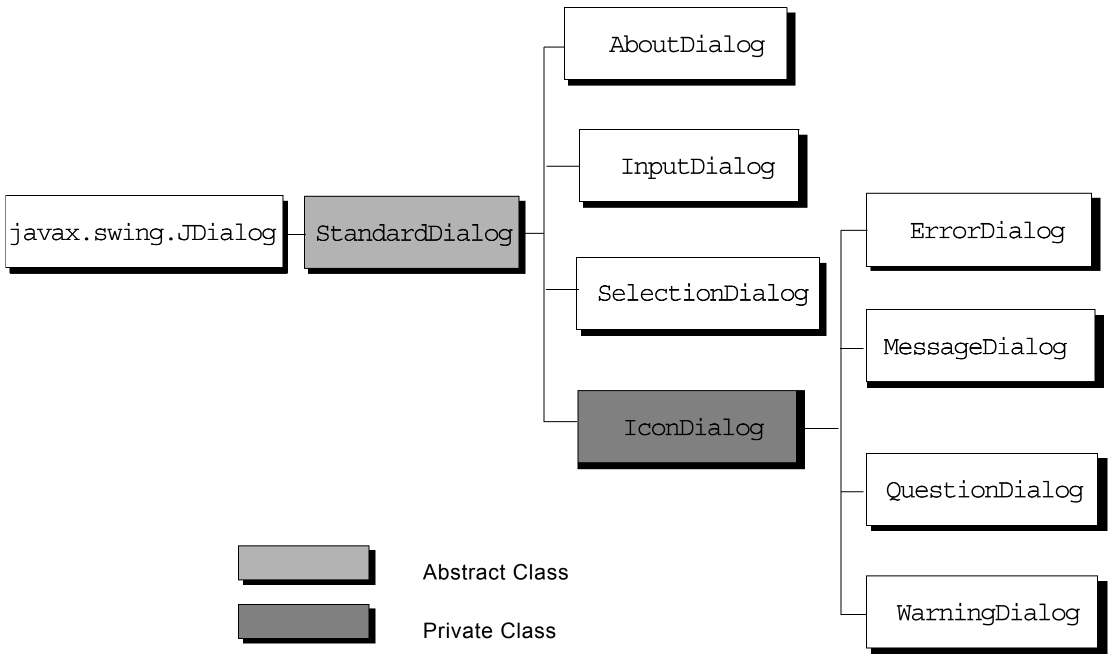
Using Standard Dialogs

This section provides the following information and techniques for using standard dialogs:

- The inheritance structure for standard dialog classes.
- Common arguments to standard dialog constructors.
- Listening for dialog events.
- Localizing standard dialog text.
- Creating and launching standard dialogs.

Inheritance Structure of the Standard Dialog Classes

All standard dialogs inherit from `javax.swing.JDialog`, as this diagram illustrates:



Common Arguments to Standard Dialog Constructors

Standard dialogs provide a set of common arguments in their public constructor, which always appear in the same order. Most standard dialogs provide one or more additional arguments as well, which always appear just before the last argument.

This table describes the common arguments to the public constructor of any standard dialog, in order:

Type and Argument	Description
`Frame parent`	The first argument is the parent `java.awt.Frame` in which the dialog is centered when it is launched. This argument can be `null`, in which case the dialog is centered in the frame returned by `getCurrentFrame` on a `com.gensym.core.UiApplication`, or centered in the screen, if the application is not a subclass of `UiApplication`.
`String title`	The second argument is the dialog title as a `java.lang.String`, which you can localize.
`boolean isModal`	The third argument is a `boolean` value that determines whether the dialog is modal. A **modal dialog** is one that the user must dismiss before performing any other action in the application.
`StandardDialogClient client`	The last argument is an instance of a `StandardDialogClient`, which gets notified when the user has clicked a dialog button. This argument can be `null` if the dialog requires no post-processing.

For information about `StandardDialogClient`, see "Listening for Dialog Events" on page 77.

Listening for Dialog Events

Any class can implement a `StandardDialogClient`, which is an interface that:

- Gets notified when the user has clicked a button on any standard dialog.
- Implements the behavior of the dialog when it is dismissed in the `dialogDismissed` method.

For example, you might want to launch a `WarningDialog` when the application is in a particular state. The class that launches the dialog would implement a `StandardDialogClient` to receive notification when the user has clicked the OK button by invoking the client's `dialogDismissed` method.

To listen for dialog events:

➔ Create a class that implements this interface:

```
com.gensym.dlg.StandardDialogClient
```

The following sections describe typical features of such an implementation.

Implementing the Behavior of the Dialog When it is Dismissed

The `StandardDialogClient` typically implements one or more of the following tasks in its `dialogDismissed` method:

- Tests whether the dialog was cancelled by calling:

  ```
  wasCancelled()
  ```

  If the method returns `true`, the method dismisses the dialog without applying the edits.

- If the dialog accepts user input, obtains the results of the dialog by calling:

  ```
  getResults()
  ```

  The `getResults` method returns a string or an array of strings, depending on the type of dialog. For example, calling `getResults` on an `InputDialog` that provides multiple text fields returns an array of strings, whereas calling `getResults` on a `SelectionDialog` that allows a single selection returns a string.

  For more information, see "InputDialog" on page 99 and "SelectionDialog" on page 106.

- Uses the results of the dialog to perform some action.

  For example, if the dialog provides Host and Port fields, the `dialogDismissed` method might use these values to connect to G2.

- If necessary, explicitly closes the dialog by calling:

  ```
  setVisible(false)
  ```

Determining Which Button the User Has Clicked

The `dialogDismissed` method takes two arguments:

- `StandardDialog d` — Any subclass of `StandardDialog`.
- `int cmdCode` — An integer that determines which dialog button the user has clicked.

The `cmdCode` argument is called a **command code**. All standard dialogs implement the following interface, which provides static final variables for use as command codes:

```
com.gensym.dlg.CommandConstants
```

For example, to determine which button the user has clicked in a `QuestionDialog`, you use the following command codes:

- `YES`
- `NO`

For an example that uses command codes to customize the buttons that appear in a dialog, see "Customizing Dialog Buttons and Icons" on page 86.

To determine which button the user has clicked:

➔ Refer to the `cmdCode` argument to the `dialogDismissed` method in the implementation of this method.

For example, the following code fragments might appear in the `dialogDismissed` method of a `StandarDialogClient` that launches a `QuestionDialog`, where `code` is the `cmdCode` argument to the `dialogDismissed` method. The action of each conditional statement depends on purpose of the dialog.

```
if (code == CommandConstants.YES) {
   //Perform the action when the user clicks YES
}

if (code == CommandConstants.NO) {
   //Perform the action when the user clicks NO
}
```

Localizing Dialog Text

You can localize these pieces of dialog text when you create a standard dialog, depending on the type of dialog:

- Title
- Text labels
- Message text

To localize dialog button text, you must create a custom dialog, described in "Customizing Dialog Buttons and Icons" on page 86.

In the examples that follow, `i18nUI` and `bundle` are instances of a `com.gensym.message.Resource`, which is a G2 JavaLink class that supports localization.

For general information about using resources, see Appendix A, "Localization."

Examples

Localizing the Title and Prompt of a SelectionDialog

You can localize the title and prompt of a SelectionDialog. In the following example, getWkspTitle and getWkspPrompt are the keys.

```
private com.gensym.message.Resource i18nUI = Resource.getBundle
   ("com.gensym.demos.wksppanel.UiLabels");
private com.gensym.core.UiApplication application;

new SelectionDialog (application.getCurrentFrame(),
                     i18nUI.getString ("getWkspTitle"),
                     false,
                     i18nUI.getString ("getWkspPrompt"),
                     names, false,
                     SelectionDialog.NO_SELECTION, true,
                     getHandler);
```

For more information, see “SelectionDialog” on page 106.

Localizing the Dialog Title and Text Field Labels

You can localize the title and text box labels of an InputDialog. In the following example, title and numberOfMoons are the keys.

```
private com.gensym.message.Resource bundle =
   Resource.getBundle("com.gensym.demos.
   internationalizationdemo.InternationalizationDemoResource");
private com.gensym.core.UiApplication application;

//Localize labels
String dialogTitle = bundle.getString("title");
String textboxLabel = bundle.getString("numberOfMoons");

//Define labels and initial values
String[] textFieldLabels = new String[]{textboxLabel};
String[] initialValues = new String[]
   {Integer.toString(numberOfMoons)};

//Create input dialog
new InputDialog(application.getCurrentFrame(), dialogTitle,
                true, textFieldLabels, initialValues,
                (StandardDialogClient)this);
```

For more information, see “InputDialog” on page 99.

Localizing Dynamically Updating Error Messages

Typically, dialog messages consist of static and dynamic text, for example:

```
Error on line 5 of MyExample.java
```

To localize dynamic message text, call the `format` method on a `com.gensym.message.Resource`. You provide a key and an object, or a key and an array of objects as arguments. The `Resource` dynamically updates the text associated with the key by substituting the object(s) for special characters in the text.

For example, this code fragment throws an exception by calling `format` on the `i18n` resource, providing a key and an object as arguments:

```
throw new IllegalStateException
   (i18n.format("CommandIsUnavailable", cmdKey));
```

Here is the command key and text as they appear in the error resource properties file, where the dialog substitutes the special sequence of characters `{0}` with the `cmdKey` argument:

```
CommandIsUnavailable=Command {0} is unavailable
```

Creating and Launching Standard Dialogs

Typically, you provide a frame as the first argument to the constructor of a standard dialog to center the dialog in the parent frame when you launch it.

Each standard dialog class takes a unique set of arguments, in addition to the common arguments described in “Common Arguments to Standard Dialog Constructors” on page 76.

For details, see “Standard Dialogs Reference” on page 94.

To launch a standard dialog:

1 Create an instance of a standard dialog by calling its constructor.

2 If your dialog needs to support specific behavior when it is dismissed, implement a `StandardDialogClient`.

   We recommend that you define an implementation of the `StandardDialogClient` as an anonymous inner class.

   For details, see “Listening for Dialog Events” on page 77.

3 Launch the dialog by calling this method:

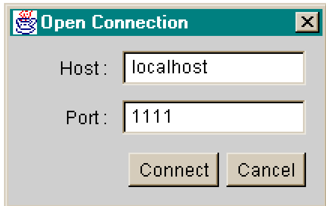
```
setVisible(true)
```

Examples

Launching an InputDialog that Connects to G2

The following example creates and launches an `InputDialog` for connecting to G2. The dialog provides a Host and Port field, which the method uses to make the connection.

Here is the dialog the example creates:



The example method performs these tasks:

- Defines an implementation of a `StandardDialogClient`, whose `dialogDismissed` method:
  - Returns if the dialog is cancelled.
  - Gets the results from the dialog and uses them to make a connection through a `com.gensym.ntw.TwGateway`.
  - Creates and launches an `ErrorDialog` if the connection fails.
- Creates and launches an `InputDialog`, passing in localized labels and initial values for the Host and Port fields.

Here is the `openConnection` method that creates and launches the dialog:

```
private static com.gensym.message.Resource i18nUI = Resource.getBundle
    ("com.gensym.demos.wksppanel.UiLabels");
private com.gensym.core.UiApplication application;

private void openConnection () {

    //Localize text and provide initial values
    String[] labels = {i18nUI.getString ("hostPrompt"),
                       i18nUI.getString ("portPrompt")};
    String[] initialValues = {"localhost", "1111"};

    //Define the StandardDialogClient
    StandardDialogClient openHandler = new StandardDialogClient () {
        public void dialogDismissed (StandardDialog d, int code) {
            try {
                InputDialog id = (InputDialog)d;

                //Return if dialog is cancelled
                if (id.wasCancelled ()) return;

                //Get the results from the dialog
                String[] results = id.getResults ();
                String host = results[0];
                String port = results[1];
```

```
                //Use the results to make a connection
                TwAccess connection = TwGateway.openConnection (host,
                                                                 port);
                connection.login();
              //Handle exceptions and launch ErrorDialog
              } catch (Exception e) {
                new ErrorDialog (null, "Error During Connect", true,
                                    e.toString (), null).setVisible (true);
              }
            }
          };
          //Create and launch an InputDialog
          new InputDialog
              (application.getCurrentFrame (),
               i18nUI.getString ("openConnectionTitle"),
               true, labels, initialValues,
               openHandler).setVisible (true);
        }
```

Launching a SelectionDialog that Gets a Named Workspace

The following example launches a `SelectionDialog` for getting a named KB workspace.

Here is the dialog the example creates:

The example method performs these tasks:

- Gets the print value of each named KB workspace by calling `getNamedWorkspaces` on a connection.
- Defines an implementation of a `StandardDialogClient`, whose `dialogDismissed` method:
  - Returns if the dialog is cancelled.
  - Initializes a variable for the `SelectionDialog`.

   - Gets the result from the dialog and uses it to create a symbol for the selected KB workspace.
   - Creates and starts a new thread to download the selected KB workspace.
- Creates and launches the dialog, providing these unique arguments to the `SelectionDialog`:
   - `String prompt` — The dialog prompt that appears above the list of options, in this case, a localized text string.
   - `String initialValues[]` — The list of available named KB workspaces, which the method obtains from the connection.
   - `boolean allowMultipleSelections` — `false`, which indicates that the user can choose one item only from the list.
   - `int initialSelection` — `NO_SELECTION`, which indicates that no item should be selected initially; otherwise, the index of the initially selected item.
   - `boolean requireSelection` — `true`, which indicates that the user must make a selection before being allowed to accept the dialog.
- Handles exceptions.

The `dialogDismissed` method creates an instance of the following inner class, which starts a new thread to download a KB workspace and add it to the application:

```
class WorkspaceDownloaderThread extends Thread {

   public void run () {

      //Get the unique named item from the connection
      //and add it to the application

   }
}
```

Here is the `getWorkspace` method that creates and launches the dialog:

```
private com.gensym.core.UiApplication application;
private com.gensym.message.Resource i18nUI = Resource.
   getBundle("com.gensym.demos.singlecxnsdiapp.UiLabels");

private void getWorkspace () {
   try {

      //Get a sequence of named workspace from connection
      final Sequence wkspNames =
         application.getConnection().getNamedWorkspaces ();

      //Determine its size
      int numWksps = wkspNames.size ();
```

```
        //Create array of strings to hold each workspace name
        String[] names = new String [numWksps];

        //Iterate through the sequence getting each wksp name
        for (int i=0; i<numWksps; i++)
           names[i] = ((Symbol)wkspNames.elementAt(i)).getPrintValue();

     //Define a StandardDiaolgClient
     StandardDialogClient getHandler = new StandardDialogClient () {
        public void dialogDismissed (StandardDialog d, int code) {
           if (d.wasCancelled ()) return;
           SelectionDialog sd = (SelectionDialog)d;
           int chosenIndex = sd.getResult ();
           Symbol wkspName_ = (Symbol) wkspNames.
              elementAt (chosenIndex);
           new WorkspaceDownloaderThread(application,
                                         wkspName_).start ();
        }
     };

     //Create and launch a SelectionDialog
     new SelectionDialog (application.getCurrentFrame(),
                          i18nUI.getString ("getWkspTitle"),
                          false, i18nUI.getString("getWkspPrompt"),
                          names, false, SelectionDialog.NO_SELECTION,
                          true, getHandler).setVisible (true);

     //Handle exceptions
     } catch (Exception e) {
        new WarningDialog (null,
                           i18nUI.getString ("getWkspError"),
                           true, e.toString (),
                           null).setVisible (true);
     }
  }
```

Customizing Dialogs

You create custom dialogs by extending the standard dialog classes:

To customize...	Extend...
The buttons, icons, or behavior of any standard dialog class	A standard dialog class, such as `InputDialog`.
The dialog controls that appear in the dialog component area	The `StandardDialog` abstract class.

The following sections describe how to customize:

- Dialog buttons and icons.
- Dialog behavior when it is launched or dismissed.
- Dialog controls.

For examples of custom dialogs, see “Example” on page 90.

For information about creating custom dialogs, see the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

Customizing Dialog Buttons and Icons

The standard dialog classes provide a protected constructor, which you use to customize:

- The dialog buttons that appear.
- The dialog button text, which you can localize.
- The alignment of the buttons in the command panel.
- The icon that appears, in dialogs that support icons.

For example, you might want to add an Apply button, which applies the edits and leaves the dialog open.

Calling the Protected Constructor

In addition to the arguments described in “Common Arguments to Standard Dialog Constructors” on page 76, the protected constructor takes these arguments:

Type and Argument	Description
`String buttonLabels[]`	An array of strings that provide labels for each button.
`int buttonCodes[]`	An array of command codes that determine how the `StandardDialogClient` interprets the button the user has clicked.
`Image img`	An instance of a `java.awt.Image` that provides the icon to display. This argument is only available to dialogs that define an icon.

Using Command Constants and Standard Dialog Constants

All standard dialogs provide the set of static final variables shown in the following table. You use the command code constants to specify which buttons appear in a dialog, you use the button label constants to localize button labels, and you use the button alignment constants to specify the alignment of the buttons in the dialog.

Command Codes	Button Labels	Button Alignment
`OK`	`OK_LABEL`	`CENTER`
`APPLY`	`APPLY_LABEL`	`LEFT`
`CANCEL`	`CANCEL_LABEL`	`RIGHT`
`DISMISS`	`DISMISS_LABEL`	
`YES`	`YES_LABEL`	
`NO`	`NO_LABEL`	
`HELP`	`HELP_LABEL`	

The following interface defines the command code constants:

```
com.gensym.dlg.CommandConstants
```

This abstract class defines the button label and button alignment constants:

```
com.gensym.dlg.StandardDialog
```

Note The `OK`, `CANCEL`, and `DISMISS` command codes automatically close the dialog when the user clicks the appropriate button.

Customizing Button Labels, Command Codes, and Icons

To customize button labels, command codes, and/or icons of a standard dialog:

1. Create a subclass of the `StandardDialog` class whose labels, command codes, and/or icon you want to customize.
2. In the constructor for the custom dialog, call the protected constructor for the superior class, specifying the button labels, command codes, and/or icon as arguments.

For example, the following code fragment defines a custom `InputDialog` class. The constructor calls the protected constructor for the superior class to provide custom button labels and button codes.

```
class ConnectionDialog extends InputDialog {
   ConnectionDialog (Frame f, String title, boolean isModal,
                     String[] prompts, String[] initValues,
                     String[] btnLabels, int[] btnCodes,
                     StandardDialogClient client) {
      super (f, title, isModal, prompts, initValues,
            btnLabels, btnCodes, client);
}
```

Customizing Button Alignment

By default, the dialog buttons are centered in the dialog's command panel, using a `java.awt.FlowLayout`.

To customize the button alignment of a standard dialog:

1. Create a subclass of the standard dialog class whose button alignment you want to customize.
2. Override the following method of the custom dialog:

   ```
   getButtonAlignment()
   ```

The method can return one of these constants:

```
CENTER
LEFT
RIGHT
```

For example, the following method left-aligns the buttons in a custom dialog:

```
protected int getButtonAlignment () {
   return LEFT;
}
```

Customizing Dialog Behavior When it is Launched or Dismissed

You can customize the behavior of a standard dialog when it is launched or dismissed. For example, you might want the dialog to play a sound when it is launched or to launch in a location other than in the center of the parent frame.

To customize the behavior when a standard dialog is launched or dismissed:

1 Create a subclass of the standard dialog class whose launch and/or dismiss behavior you want to customize.

2 Override the following method of the custom dialog:

```
setVisible (boolean showQ)
```

For example, this method overrides the behavior of a custom dialog so it beeps when it is launched:

```
public void setVisible (boolean showQ) {
    if (showQ == true)
        java.awt.Toolkit.getDefaultToolkit().beep ();
    super.setVisible(showQ);
}
```

Customizing Dialog Controls

You can create a custom dialog with different types of controls, such as text fields and radio buttons.

To customize dialog controls:

1 Create a class that extends:

```
com.gensym.dlg.StandardDialog
```

2 Call the constructor for the superior class in the custom dialog's constructor.

3 Create the Java component to display.

The following sections explain steps 2 and 3 in detail.

Calling the Constructor for StandardDialog

In the constructor for the subclass, call the constructor for `StandardDialog` with these arguments:

- `Frame parent` — The parent frame in which to center the dialog.
- `String title` — The dialog title.
- `boolean isModal` — `true` to launch the dialog modally, `false` otherwise.
- `String buttonNames[]` — An array of strings that represent the button labels.

- `int cmdCodes[]` — An array of integers that represent the command codes for each button.
- `Component x` — The Java component to add to the center of the dialog.
- `StandardDialogClient client` — The client that specifies the behavior when the dialog is dismissed, or `null`.

For example, the following code fragment calls the constructor for `StandardDialog` by referring to the static final variables that define button labels and command codes. The component that gets added is `PumpPanel`, which is a `java.awt.Panel`.

```
super (parent, title, isModal,
       new String[] {OK_LABEL, APPLY_LABEL, CANCEL_LABEL},
       new int[] {OK, APPLY, CANCEL},
       new PumpPanel(), client);
```

Creating the Component to Display in the Dialog

The component to display can be:

- An individual Java component.
- A Java container with multiple Java components and an associated layout manager.

To create the component to display in the dialog:

1. Create an instance of any `java.awt.Component`, such as a `java.awt.Panel`.
2. Add the desired control(s) to the component, such as text areas, text fields, radio boxes, and/or choice boxes.
3. To ensure the dialog looks good on all platforms, arrange the controls, using a layout manager, such as a `java.awt.GridBagLayout`.

Example

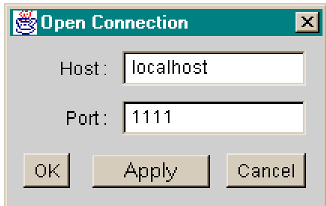
Creating a Custom InputDialog with OK, Apply, and Cancel Buttons

This example creates and launches a custom dialog called `ConnectionDialog`, which provides two G2 text fields for entering the Host and Port of a connection to G2.

`ConnectionDialog` provides these custom dialog features:

- OK, Apply, and Cancel buttons.
- Dialog buttons that are centered in the command panel.
- A beep when the dialog makes the connection.

The custom dialog looks like this:



The example method defines an implementation of a `StandardDialogClient` so the command is notified when the user has clicked a dialog button.

The implementation of the `dialogDismissed` method checks to see which dialog button the user has clicked, as follows:

- If the user clicks the Apply button, the client gets the host and port from the dialog and opens a connection.
- If the user clicks the OK button, the dialog performs the same actions as if the user clicked the Apply button, plus it closes the dialog.

The method creates an instance of `ConnectionDialog`, which is an inner class. `ConnectionDialog` extends `InputDialog` and overrides the dialog buttons, button alignment, and dismiss behavior.

The `ConnectionDialog` uses localized text labels, button labels, and dialog title by providing keys and a resource bundle.

Here is the `openConnection` method, which creates and launches a custom `InputDialog` for connecting to G2:

```
private static com.gensym.message.Resource i18nUI = Resource.getBundle
    ("com.gensym.demos.wksppanel.UiLabels");
private com.gensym.core.UiApplication application;

private void openConnection () {

    //Initialize variables for creating the custom dialog
    String[] labels = {i18nUI.getString ("hostPrompt"),
                       i18nUI.getString ("portPrompt")};
    String[] initialValues = {"localhost", "1111"};
    String[] buttonLabels = {i18nUI.getString ("okLabel"),
                             i18nUI.getString ("applyLabel"),
                             i18nUI.getString ("cancelLabel")};
    int[] buttonCodes = {com.gensym.dlg.CommandConstants.OK,
                         com.gensym.dlg.CommandConstants.APPLY,
                         com.gensym.dlg.CommandConstants.CANCEL};
```

```
    //Implement a StandardDialogClient to open a connection
    StandardDialogClient openHandler =
       new StandardDialogClient () {
          public void dialogDismissed (StandardDialog d, int code) {
             try {
                InputDialog id = (InputDialog)d;
                if (id.wasCancelled ()) return;
                String[] results = id.getResults ();
                String host = results[0];
                String port = results[1];

                //If the user clicks Apply or OK,
                //connect and beep
                if (code == CommandConstants.APPLY ||
                   code == CommandConstants.OK) {
                   TwAccess connection =
                      TwGateway.openConnection (host, port);
                   connection.login();
                   application.setConnection (connection);
                   java.awt.Toolkit.getDefaultToolkit().beep();
                }
             } catch (Exception e) {
                new ErrorDialog (null, "Error During Connect",
                   true, e.toString (), null).setVisible (true);
             }
          }
    };

    //Create an instance of a ConnectionDialog
    new ConnectionDialog
       (application.getCurrentFrame(),
        i18nUI.getString ("openConnectionTitle"),
        true, labels, initialValues, buttonLabels, buttonCodes,
        (StandardDialogClient) openHandler).setVisible (true);
}

//Define an inner class for ConnectionDialog
class ConnectionDialog extends InputDialog {
    ConnectionDialog (Frame f, String title, boolean isModal,
                      String[] prompts, String[] initValues,
                      String[] btnLabels, int[] btnCodes,
                      StandardDialogClient client) {
       super (f, title, isModal, prompts, initValues,
             btnLabels, btnCodes, client);
    }

    //Override button alignment from super class
    protected int getButtonAlignment () {
       return CENTER;
    }
```

```
    //Override behavior when dialog closes
    public void setVisible (boolean showQ) {
        if (showQ == false)
        java.awt.Toolkit.getDefaultToolkit().beep ();
        super.setVisible(showQ);
    }
}
```

Standard Dialogs Reference

The following sections provide reference information for each standard dialog class. Each reference section provides:

- A sample dialog.
- A description.
- The constructor or constructors, and the unique arguments to the public constructor.
- An example.

Each dialog inherits its definition from this class:

```
com.gensym.dlg.StandardDialog
```

For general information on standard dialogs, see:

- "Using Standard Dialogs" on page 75.
- "Customizing Dialogs" on page 85.

The two categories of standard dialogs are:

- Dialogs that accept user input.
- Informational dialogs.

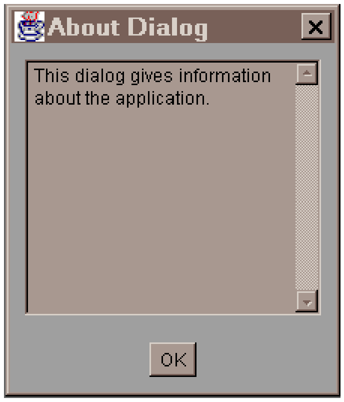
User Input Dialogs

```
InputDialog
QuestionDialog
SelectionDialog
```

Informational Dialogs

```
AboutDialog
ErrorDialog
MessageDialog
WarningDialog
```

AboutDialog



Description

`AboutDialog` defines a single `java.awt.TextArea`, with or without scroll bars, in which to display help text. It provides an OK button to dismiss the dialog.

For information on...	See...
Localizing help text	"Localizing Dialog Text" on page 79.
Customizing the dialog button	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

The public `AboutDialog` constructor takes the following arguments, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String aboutString`	The text string to display in the dialog's text area, which can include any Java escape characters.
`int numRows`	The number of rows in the text area.

Type and Argument	Description
int numColumns	The number of columns in the text area.
int scrollbarVisibility	A variable that determines whether the text area contains vertical and/or horizontal scroll bars, where the options include any of the static final variables defined on java.awt.TextArea.

For a description of the common arguments to all standard dialogs, see "Common Arguments to Standard Dialog Constructors" on page 76.

Example

The method in this example creates and launches an AboutDialog. The dialog has vertical scroll bars in the text area. The dialog uses localized help text and a localized title by providing keys and a resource bundle.

The constructor passes null as the StandardDialogClient argument because no follow-up action is required.

Here is the method that creates and launches the dialog, with the constructor shown in bold:

```
private com.gensym.message.Resource i18nUI = Resource.getBundle
    ("com.gensym.demos.singlecxnmdiapp.UiLabels");
private java.awt.Frame frame;

private void handleAboutApplication(){
    if (aboutDialog == null){
        String title = i18nUI.getString("AboutTitle");
        String msg = i18nUI.getString("AboutMessage");
        boolean isModal = true;
        int numRows = 25;
        int numColumns = 80;
        int scrollbarVisibility = TextArea.SCROLLBARS_VERTICAL_ONLY;
        aboutDialog = new AboutDialog(frame, title, isModal,
                                      msg, numRows, numColumns,
                                      scrollbarVisibility,
                                      null);
    }
    aboutDialog.setVisible (true);
}
```

ErrorDialog

Description

ErrorDialog provides an error message, an icon, and an OK button for acknowledging the error.

If the dialog is launched modally, it beeps when it is launched.

For information on...	See...
Launching dialogs modally	"Common Arguments to Standard Dialog Constructors" on page 76.
Localizing error text	"Localizing Dialog Text" on page 79.
Customizing the button and icon	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

ErrorDialog provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	The default icon and OK button for dismissing the dialog.
The protected constructor	A custom icon and/or button.

The public `ErrorDialog` constructor takes the following argument, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String message`	The error message to display in the dialog, which can include any Java escape characters.

For a description of the common arguments to all standard dialogs, see "Common Arguments to Standard Dialog Constructors" on page 76.

For information on the standard arguments that the protected constructor provides, see "Calling the Protected Constructor" on page 86.

Example

The following method creates and launches an `ErrorDialog`, with the constructor shown in bold.

The constructor passes `null` as the `StandardDialogClient` argument because no follow-up action is required.

```
public void handleErrorDialog(){
    if (errorDialog == null){
      boolean isModal = false;
      errorDialog = new ErrorDialog(null, "Error Dialog",
                                    isModal,
                                    "This is the error message!",
                                    null);
    }
    errorDialog.setVisible(true);
}
```

InputDialog

Description

InputDialog creates a dialog with one or more instances of a java.awt.TextField. You provide the prompts and initial values for each text field in the constructor.

An InputDialog provides an OK button for accepting the user input, and a Cancel button for discarding the input and closing the dialog.

You get the results of the editing session by calling getResults on the dialog after it is dismissed. This method returns an array of strings, where each string represents the value of each text field.

For information on...	See...
Localizing dialog text	"Localizing Dialog Text" on page 79.
Customizing the buttons	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

InputDialog provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	Text fields, initial values, and the OK and Cancel buttons.
The protected constructor	Custom dialog buttons.

The public `InputDialog` constructor takes these arguments, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String labels[]`	An array of strings that provide the labels for each text field.
`String initialValues[]`	An array of strings that provide the initial values for each text field.

For a description of the common arguments to all standard dialogs, see “Common Arguments to Standard Dialog Constructors” on page 76.

For information on the standard arguments to the protected constructor, see “Calling the Protected Constructor” on page 86.

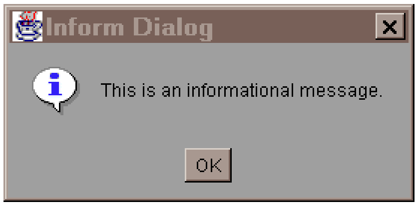
Example

This example method creates and launches an `InputDialog` for connecting to G2. The method creates a `StandardDialogClient` to receive notification when the user has clicked a dialog button. The implementation of the `dialogDismissed` method gets the results from the dialog, makes a connection, and makes a login request to a secure G2.

Here is the openConnection method, with the constructor shown in bold:

```
private static com.gensym.message.Resource i18nUI = Resource.
   getBundle ("com.gensym.demos.singlecxnsdiapp.UiLabels");
private java.awt.Frame frame;

private void openConnection () {
   String[] labels = {i18nUI.getString ("hostPrompt"),
                      i18nUI.getString ("portPrompt")};
   String[] initialValues = {"localhost", "1111"};
   StandardDialogClient openHandler = new StandardDialogClient() {
      public void dialogDismissed (StandardDialog d, int code) {
         try {
            InputDialog id = (InputDialog)d;
            if (id.wasCancelled ()) return;
            String[] results = id.getResults ();
            String host = results[0];
            String port = results[1];
            TwAccess connection = TwGateway.openConnection(host,
                                                          port);
            connection.login();
         } catch (Exception e) {
            new ErrorDialog (null, Error During Connect",
                             true, e.toString (),
                             null).setVisible (true);
         }
      }
   };
   new InputDialog
       (frame, i18nUI.getString ("openConnectionTitle"),
        true, labels, initialValues, openHandler).setVisible
(true);
}
```

MessageDialog



Description

`MessageDialog` provides an informational message and an OK button for acknowledging the message.

If the dialog is launched modally, it beeps when it is launched.

For information on...	See...
Launching dialogs modally	"Common Arguments to Standard Dialog Constructors" on page 76.
Localizing message text	"Localizing Dialog Text" on page 79.
Customizing the button and icon	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

`MessageDialog` provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	The default icon and OK button for dismissing the dialog.
The protected constructor	A custom icon and/or button.

The public `MessageDialog` constructor takes this argument, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String message`	The message to display in the dialog, which can include any Java escape characters.

For a description of the common arguments to all standard dialogs, see "Common Arguments to Standard Dialog Constructors" on page 76.

For information on the standard arguments that the protected constructor provides, see "Calling the Protected Constructor" on page 86.

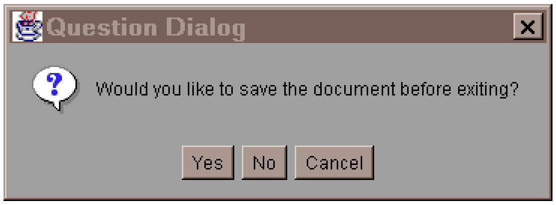
Example

The constructor passes `null` as the `StandardDialogClient` argument because no follow-up action is required.

The following method creates and launches a `MessageDialog`, with the constructor shown in bold.

```
private java.awt.Frame frame;

public void handleMessageDialog(){
   if (messageDialog == null){
      boolean isModal = false;
      messageDialog = new MessageDialog
         (frame, "Message Dialog", isModal,
          "This is a short message.", null);
   }
   messageDialog.setVisible(true);
}
```

QuestionDialog



Description

`QuestionDialog` provides a question that the user must answer by clicking the Yes or No button.

For information on...	See...
Implementing the behavior of the dialog when the user has clicked each button	"Determining Which Button the User Has Clicked" on page 78.
Localizing question text	"Localizing Dialog Text" on page 79.
Customizing the buttons and icon	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

`QuestionDialog` provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	The default icon, and the Yes, No, and Cancel buttons.
The protected constructor	A custom icon and/or buttons.

The public `QuestionDialog` constructor takes this argument, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String message`	The question to display in the dialog, which can include any Java escape characters.

For a description of the common arguments to all standard dialogs, see "Common Arguments to Standard Dialog Constructors" on page 76.

For information on the standard arguments that the protected constructor provides, see "Calling the Protected Constructor" on page 86.

Example

The following example creates and launches a `QuestionDialog`, with the constructor shown in bold.

The constructor passes `this` as the `StandardDialogClient` argument, which is an implementation of `StandardDialogClient`.

The implementation of the `dialogDismissed` method tests its command code argument to determine which button the user has clicked, and prints different messages to the standard output window.

```
private java.awt.Frame frame;

public void handleQuestionDialog(){
   if (questionDialog == null){
      boolean isModal = false;
      questionDialog = new QuestionDialog
         (frame, "Question Dialog", isModal,
         "Would you like to save before exiting?", this);
   }
   questionDialog.setVisible(true);
}

public void dialogDismissed (StandardDialog dlg,
                             int code) {
   if (dlg instanceof QuestionDialog){
      if (code == YES)
         System.out.println("Save before exiting");
      else
         System.out.println("Do not save before exiting");
}
```

SelectionDialog

Description

`SelectionDialog` provides a list of items from which to choose one or more items. You provide these items in the constructor:

- The prompt.
- The list of items from which to choose.
- Whether the dialog supports a single selection or multiple selections.
- The initially selected item.
- Whether the dialog requires a selection before the user is allowed to dismiss it.

You get the result from the dialog by calling one of these two methods, depending on whether the dialog supports single or multiple selections:

- `getResult` — In a dialog that supports a single selection, returns a string that is the selected item.
- `getResults` — In a dialog that supports multiple selections, returns an array of strings representing the value of each selected item.

For information on...	**See...**
Localizing dialog text	“Localizing Dialog Text” on page 79.
Customizing the buttons	“Customizing Dialog Buttons and Icons” on page 86.
Customizing the dialog behavior	“Customizing Dialog Behavior When it is Launched or Dismissed” on page 89.

Constructor

`SelectionDialog` provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	A scrollable selection list, and the OK and Cancel buttons.
The protected constructor	Custom dialog buttons.

The public `SelectionDialog` constructor takes these arguments, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String prompt`	The prompt string to display above the list of items.
`String initialValues[]`	An array of strings that specify the items in the selection.
`boolean allowMultipleSelection`	A `boolean` that determines whether the dialog allows the user to select multiple items.
`int initialSelection`	An integer that specifies the item that is selected by default. Use the `NO_SELECTION` static final variable if no initial selection exists.
`boolean requireSelection`	A `boolean` that determines whether the user must select an item before being allowed to accept the dialog.

For a description of the common arguments to all standard dialogs, see “Common Arguments to Standard Dialog Constructors” on page 76.

For information on the standard arguments that the protected constructor provides, see “Calling the Protected Constructor” on page 86.

Example

The following example creates and launches a `SelectionDialog` for choosing among several strings, with the constructor shown in bold.

The method uses this private variable:

```
private java.awt.Frame frame;
```

The implementation of the `dialogDismissed` method prints the selected value to the standard output window.

```
public void handleSelectionDialog(){
   if (selectionDialog == null){
      boolean isModal = false;
      String[] selectionDialogInitialValues  =
         {"Apples", "Bananas", "Grapes", "Oranges",
          "Pears", "Peaches"};
      boolean allowMultiSelect = false;
      int initialSelection = 2;
      boolean requireSelection = false;
      selectionDialog =
         new SelectionDialog (frame, "Selection Dialog",
                              isModal, "Selection Prompt:",
                              selectionDialogInitialValues,
                              allowMultiSelect,
                              initialSelection,
                              requireSelection, this);
   }
   selectionDialog.setVisible(true);
}

public void dialogDismissed (StandardDialog dlg, int code) {
   if (dlg instanceof SelectionDialog){
      SelectionDialog selectionDlg = (SelectionDialog)dlg;
      int result = selectionDlg.getResult();
      System.out.println("Selected:
         "+selectionDialogInitialValues[result]);
   }
}
```

WarningDialog

Description

`WarningDialog` displays a warning message to the user with an OK button for acknowledging the warning.

If the dialog is launched modally, it beeps when it is launched.

For information on...	See...
Launching dialogs modally	"Common Arguments to Standard Dialog Constructors" on page 76.
Localizing message text	"Localizing Dialog Text" on page 79.
Customizing the button and icon	"Customizing Dialog Buttons and Icons" on page 86.
Customizing the dialog behavior	"Customizing Dialog Behavior When it is Launched or Dismissed" on page 89.

Constructor

`WarningDialog` provides two constructors:

Use this constructor...	To create a dialog that uses...
The public constructor	The default icon, and the OK button for dismissing the dialog.
The protected constructor	A custom icon and/or button.

The public `WarningDialog` constructor takes this argument, in addition to the arguments common to all standard dialogs:

Type and Argument	Description
`String message`	The warning message to display in the dialog, which can include any Java escape characters.

For a description of the common arguments to all standard dialogs, see "Common Arguments to Standard Dialog Constructors" on page 76.

For information on the standard arguments that the protected constructor provides, see "Calling the Protected Constructor" on page 86.

Example

This example shows how you would launch a `WarningDialog` when you catch a `com.gensym.jgi.G2AccessException`. The dialog uses the exception text if it exists; otherwise, it provides localized message text, which it formats by providing a key and a resource bundle.

The `AccessError` and `AccessErrorWithReason` keys appear as follows in the resource properties file, where the additional argument to the `format` method replaces the array in the localized text string associated with the key:

```
AccessError=Error accessing connection {0}.
```

```
AccessErrorWithReason=Error accessing connection {0}.\n{1}
```

For more information on localizing and formatting message text, see "Localizing Dialog Text" on page 79.

The constructor passes `null` as the `StandardDialogClient` argument because no follow-up action is required.

Here is the statement that catches the exception, with the constructors shown in bold:

```
private com.gensym.ntw.TwGateway currentConnection;
private com.gensym.message.Resource i18n =
   Resource.getBundle("com.gensym.demos.test.ErrorResources");

catch(G2AccessException ex){
   ex.printStackTrace();
   String cxnString = currentConnection.toShortString();
   String msg = ex.getMessage();
   if (msg == null)
      new WarningDialog
          (null, i18n.getString("Error"),
           true, i18n.format("AccessError",
           cxnString), null).setVisible(true);
   else
      new WarningDialog
          (null, i18n.getString("Error"), true,
           i18n.format("AccessErrorWithReason", cxnString,
           msg), null).setVisible(true);
}
```

Creating Menus and Toolbars

Describes how to create menu bars, pulldown menus, popup menus, submenus, command groups, and toolbars from commands.

Introduction

Most modern user interfaces provide multiple ways of performing the same user action, typically through a menu bar, popup menu, and/or toolbar. While beneficial to the end user of an application, implementing these can be challenging for several reasons:

- Every view of the same command needs to remain synchronized with every other view when the status of the application changes.
- The application should not duplicate code for each view of the same action.
- Each action's view should be kept separate from its implementation so the implementation can change without requiring the views to change as well.

The `com.gensym.ui` package and its subpackages offer one solution to these problems by providing:

- A set of interfaces and classes for creating user actions through the UI.
- A set of command-aware container classes.

These classes represent a powerful way of building user interfaces without requiring the UI developer to keep track of individual instances of individual actions in individual containers.

Commands

A **command** is an action that the user can perform through the UI. A command is separate from the user interface that represents it.

The following interface defines a command:

```
com.gensym.ui.Command
```

The `Command` interface is an extension of `java.awt.event.ActionListener`, which means it defines the `actionPerformed` method to describe its action. A command receives an `ActionEvent` whenever the user invokes the action of the command by clicking a menu choice or toolbar button.

The `Command` interface is analogous to the `javax.swing.Action` interface in that it defines properties for the command's textual and iconic descriptions, its state, and its availability.

A command may perform one or more actions. You represent each action with a unique **command key**, which is a string.

You can register a client as a `CommandListener` to receive notification of `CommandEvents`, which the command delivers when the description, state, or availability of the command changes.

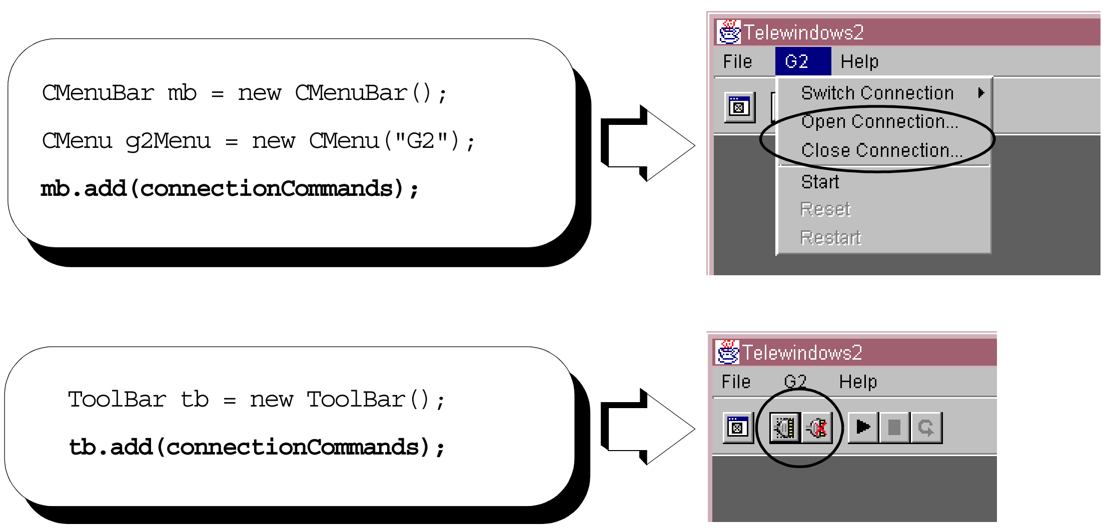
Command-Aware Containers

A **command-aware container** is a UI container that knows how to add commands, using a version of the `add` method. Command-aware containers are listeners for `CommandEvents`, which the command generates when its description, state, or availability change.

When you add a command to a command-aware container, the container:

- Represents the command appropriately for the particular container, for example, a menu creates a menu item, whereas a toolbar creates a button.
- Configures the representation based on information obtained from the command, for example:
  - Sets the text and/or icon from the command description.
  - Sets the initial state and availability.
  - Enables and disables the command when it becomes available or unavailable.
  - Sets the current state when the command state changes.
- Registers a listener with the command for notification of command events.

The following figure illustrates the result of creating a menu and a toolbar from a command:



For information on adding commands to command-aware containers, see "Creating Command-Aware Containers" on page 122.

Command-Aware Containers Based on Java Foundation Classes

Telewindows2 (TW2) Toolkit provides these command-aware containers, which are subclasses of these `javax.swing` classes:

This class...	Is a subclass of this class...
`com.gensym.ui.menu.CMenu`	`JMenu`
`com.gensym.ui.toolbar.ToolBar`	`JToolBar`
`com.gensym.ui.menu.CMenuBar`	`JMenuBar`
`com.gensym.ui.menu.CPopupMenu`	`JPopupMenu`

Command-Aware Containers Based on Java AWT Classes

TW2 Toolkit provides the following analogous menu classes, which are subclasses of these `java.awt` classes:

This class...	Is a subclass of this class...
`com.gensym.ui.menu.awt.CMenu`	`Menu`
`com.gensym.ui.menu.awt.CMenuBar`	`MenuBar`
`com.gensym.ui.menu.awt.CPopupMenu`	`PopupMenu`

Representation Constraints

You can add a command to a command-aware container as text only, icon only, or both by creating an instance of this class:

```
com.gensym.ui.RepresentationConstraints
```

`RepresentationConstraints` expose the alignment and position features supported by `javax.swing` menus and buttons, which you use to represent a command as both text and icon.

For information on adding commands to command-aware containers with constraints, see “Adding Commands with Representation Constraints” on page 127.

Structured Commands

A **structured command** is a set of related actions with a hierarchical structure and/or grouping, such as might appear in a menu with a cascading submenu. The contents of a structured command can update dynamically.

The following interface defines a structured command:

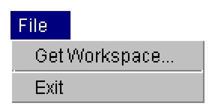
```
com.gensym.ui.StructuredCommand
```

A structured command is just like a `Command`, except that it provides additional methods that support the structure. In all other ways, structured commands are commands in the general sense of the term described earlier.

You can register a client as a `StructuredCommandListener` to receive notification of `StructuredCommandEvents`, which the command delivers when the structure changes.

Abstract Commands

TW2 Toolkit provides two default implementations of commands, which are collectively known as **abstract commands**. This table describes the abstract command classes and the interfaces they implement, and shows an example of each when added to a pulldown menu:

This class...	Implements this interface...	Which looks like this...
`com.gensym.ui.AbstractCommand`	`com.gensym.ui.Command`	
`com.gensym.ui.AbstractStructuredCommand`	`com.gensym.ui.StructuredCommand`	

By subclassing one of these abstract command classes, your command supports these features:

- Automatically notifies listeners of command events.
- Supports accessor methods for command properties.
- Supports localization of textual descriptions.

The constructor for an abstract command takes an array of objects of this class, where each object describes a single action:

```
com.gensym.ui.CommandInformation
```

A `CommandInformation` object provides the command key, the initial state and availability, a mnemonic and shortcut, the names of translation, image, and/or mnemonic resource files, and whether or not the action is immediate.

TW2 Toolkit provides several subclasses of `CommandInformation` for use with structured commands:

```
com.gensym.ui.StructuredCommandInformation
com.gensym.ui.CommandGroupInformation
com.gensym.ui.SubCommandInformation
```


For information on...	See...
Creating abstract commands	page 131
Creating abstract structured commands	page 144
Implementing the `Command` interface	page 158

Using Commands in Applications

Commands can interact with classes in an application by:

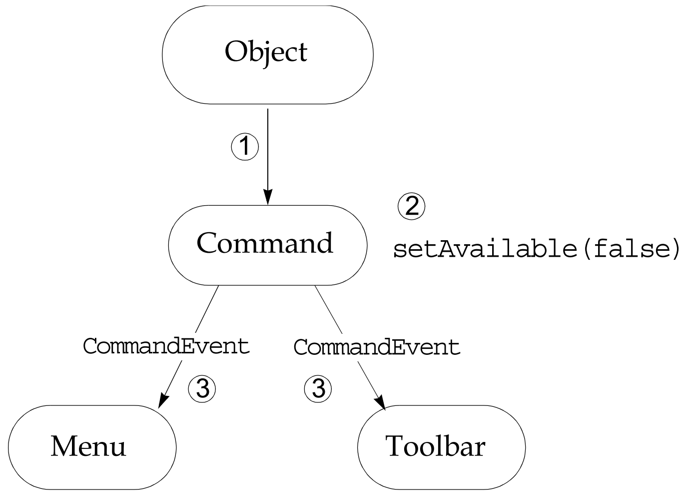
- Listening for application events and delivering command events.
- Receiving action events from command-aware containers.
- Allowing the execution of methods on application objects.

Listening for Application Events and Delivering Command Events

Your command might be a listener for some kind of application event, such as connecting to G2. In response to that event, your command might set one of its properties, such as its availability, by calling one of its `set` methods. As a result, the representation of the command in a command-aware container might become unavailable when the connection to G2 closes.

When you set a command property based on an application event, the command generates a `CommandEvent` to notify all registered listeners that the property has changed. Because command-aware containers are `CommandListeners`, they update the representation of the command in the container.

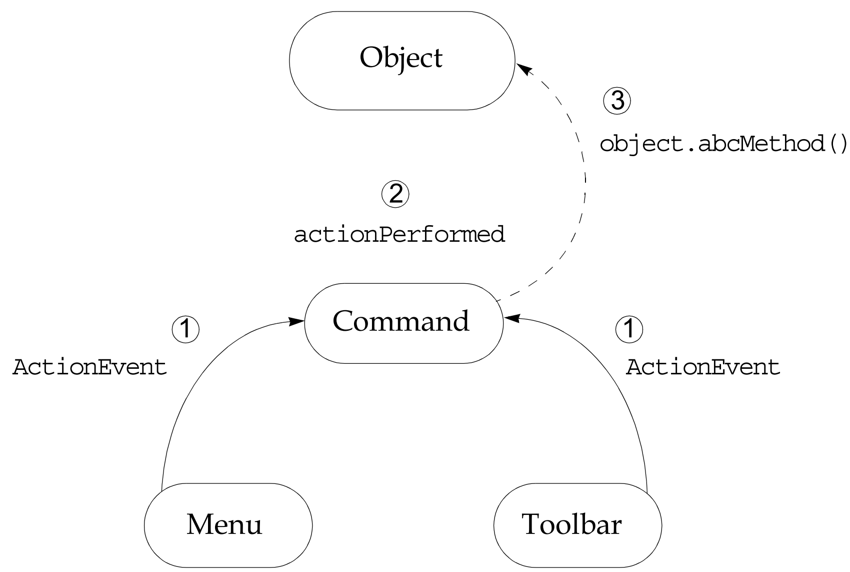
The following figure illustrates this process:



1. An application event occurs.
2. The command sets one of its properties, for example, `setAvailable(false)`.
3. The command notifies listeners that a property has changed by sending a `CommandEvent`.

Receiving Action Events From Command-Aware Containers

When the user executes the action of the command in a command-aware container, the container generates a `java.awt.event.ActionEvent`. Because commands are `ActionListeners`, they receive notification of these events and invoke their `actionPerformed` method to trigger the action, which might call a method on an object in the application. This figure illustrates this process:



1 When the user invokes a command in a command-aware container, the container notifies listeners by sending an `ActionEvent`.

2 Because a command is an `ActionListener`, it invokes its `actionPeformed` method.

3 The action of the command typically calls a method on an object in the application.

Packages Covered

com.gensym.ui

Interfaces

`Command`
`CommandListener`
`KeyableCommand`
`StructuredCommand`
`StructuredCommandListener`

Classes

`AbstractCommand`
`AbstractStructuredCommand`
`CommandEvent`
`CommandGroupInformation`
`CommandInformation`
`CommandUtilities`
`KeyInformation`
`RepresentationConstraints`
`StructuredCommandInformation`
`SubCommandInformation`

com.gensym.ui.menu

Classes

`CMenu`
`CMenuBar`
`CPopupMenu`

com.gensym.ui.menu.awt

Classes

`Menu`
`MenuBar`
`PopupMenu`

com.gensym.ui.toolbar

Classes

`ToolBar`

Relevant Demos

The following demos create TW2 Toolkit menus and toolbars from commands:

- `wksppanel`
- `singlecxnsdiapp`
- `singlecxnmdiapp`
- `multiplecxnsdiapp`
- `multiplecxnmdiapp`

The demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/`

Creating Command-Aware Containers

To create a command-aware container, you:

- Create an instance of a:
  - Menu
  - Menu bar
  - Popup menu
  - Toolbar
- Add commands to the container by adding:
  - All command keys.
  - Individual command keys.
  - All command keys or a single command key with representation constraints.
- If your container includes logical groupings of commands, add a separator, as needed.

Creating an Instance of a Command-Aware Container

By default, command-aware containers represent commands as follows:

This command-aware container...	Represents commands by using...
Menus	The short description as the menu label.
Toolbars	The small or large icon as a button, depending on whether small or large icons are in use.
Toolbars	The long description as a tool tip.

To create an instance of a `CMenu` or `CPopupMenu`, provide the title string as the argument to the constructor. To localize the menu title, use a resource and a key to provide a localized text string.

For information about localizing menu text, see Appendix A, “Localization” on page 331.

To create an instance of a `CMenuBar` or `ToolBar`, you provide no argument.

You can create menus and toolbars based on classes in the `javax.swing` package or based on classes in the `java.awt` package.

To create a command-aware container:

➔ Call the constructor for one of these classes:

```
com.gensym.ui.menu.CMenu
com.gensym.ui.menu.CPopupMenu
com.gensym.ui.menu.CMenuBar
com.gensym.ui.toolbar.ToolBar
```

or

```
com.gensym.ui.menu.awt.CMenu
com.gensym.ui.menu.awt.CPopupMenu
com.gensym.ui.menu.awt.CMenuBar
```

For example, these code fragments create instances of a `CMenu` and a `CPopupMenu`, providing a localized text string as the menu title:

```
private com.gensym.message.Resource bundle =
   Resource.getBundle("com.gensym.shell.Messages")

CMenu menu = new CMenu(bundle.getString("G2Menu"));

CPopupMenu pm = new CPopupMenu(bundle.getString("PopupTitle"));
```

These code fragments create instances of a `CMenuBar` and a `ToolBar`:

```
CMenuBar mb = new CMenuBar();
```

```
ToolBar tb = new ToolBar();
```

Adding All Command Keys

The simplest way to add a command to a command-aware container is to add all the command keys.

When creating a menu bar, you typically add instances of either of the following types of objects to create different types of menus:

- To create a simple pulldown menu, add a `CMenu`.
- To create a pulldown menu with a structure, add an implementation of the `StructuredCommand` interface.

To add a command with all its keys to a command-aware container:

➔ Call this version of the `add` method on the command-aware container:

```
add(Command cmd)
```

The argument to the `add` method is an instance of an implementation of the `Command` interface, such as a subclass of `AbstractCommand`.

Examples

Adding Commands to a Menu

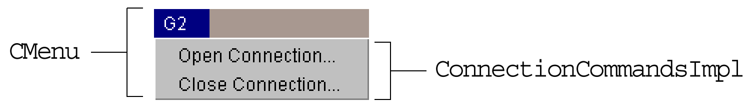
The following method creates a G2 menu, which consists of two commands. The method performs these tasks in this order:

- Creates an instance of a `CMenu`, providing a localized text string as the title.
- Adds an implementation of the `Command` interface as a handler to the menu.
- Returns the menu.

Here is the method that creates the G2 menu from commands, where `this` refers to the application:

```
private com.gensym.message.Resource bundle =
   Resource.getBundle("com.gensym.shell.Messages")

private CMenu createG2Menu() {
   CMenu menu = new CMenu(bundle.getString("G2Menu"));
   connectionHandler = new ConnectionCommandsImpl(this);
   menu.add(connectionHandler);
   return menu;
}
```

The menu looks like this:



If the command specifies a short resource file, each command key uses its short description as the menu label.

Adding Menus to a Menu Bar

You can add one or more instances of the following classes to a `CMenuBar`:

- `CMenu`, which creates a pulldown menu of commands.
- `AbstractStructuredCommand`, which creates a pulldown menu of commands with a structure.

For an example of adding an `AbstractStructuredCommand` to a `CMenuBar`, see “Examples” on page 148.

The following method:

- Creates an instance of a `CMenuBar`.
- Adds three pulldown menus, where each `create` method returns an instance of a `CMenu`.
- Returns the menu.

For the code used to implement the `createG2Menu`, see the example under “Adding All Command Keys” on page 124.

Here is the method that creates a menu bar from menus:

```
private CMenuBar createMenuBar() {
   CMenuBar mb = new CMenuBar();
   mb.add(createFileMenu());
   mb.add(createG2Menu());
   mb.add(createHelpMenu());
   return mb;
   }
```

The menu bar looks like this:

Adding Individual Command Keys

If your command defines multiple command keys, you might choose to include only certain command keys in the container. For example, a toolbar might support only certain command keys as icons, whereas a menu might support all command keys.

To add an individual command key to a command-aware container:

➔ Call this version of the add method on the command-aware container:

```
add(Command cmd, String cmdKey)
```

The cmd argument is the same as described in "Adding All Command Keys" on page 124.

The cmdKey argument is the first argument to a CommandInformation object that you pass to the constructor of an AbstractCommand.

For details, see "Implementing the Constructor" on page 146.

Example

Adding an Individual Command Key to a Toolbar

The following method creates a toolbar that consists of a single toolbar button. The method performs these tasks in this order:

- Creates an instance of a ToolBar.
- Adds the command with a single command key to the toolbar, where GET_WORKSPACE is a final static variable on the command.
- Returns the toolbar.

Here is the method that creates a toolbar from a single command key:

```
private java.awt.Frame frame;
private com.gensym.ntw.TwGateway connection;

private ToolBar createToolBar() {
   ToolBar tb = new ToolBar();
   wkspHandler = new WorkspaceCommandsImpl(frame, connection);
   tb.add(wkspHandler, WorkspaceCommandsImpl.GET_KBWORKSPACE);
   return tb;
}
```

Assuming the command defines a long resource properties file, the toolbar with its tool tip might look like this:

For more information on how to use long resource files, see "Localizing Command Text and Mnemonics" on page 138.

Adding Commands with Representation Constraints

When you add a command to a command-aware container, you can choose to use representation constraints to add the command as:

- Text only.
- Icon only.
- Icon and text.

This means, for example, you can override the default representation of a command in a particular type of container, and you can represent a command in a menu as both text and an icon.

If you choose to represent a command as both text and an icon, you can also specify the vertical and horizontal alignment, and the position of the text relative to the icon.

To add a command with representation constraints:

1. Create an instance of this class:

   ```
   com.gensym.ui.RepresentationConstraints
   ```

2. Specify the first argument to the constructor as one of the following final static variables:

   ```
   ICON_ONLY
   TEXT_ONLY
   TEXT_AND_ICON
   ```

3. If you choose to represent the command as both text and icon, optionally specify these additional arguments, in this order, in the `RepresentationConstraints` constructor:

   - `int horizontalAlignment` — Horizontal alignment of the text and icon relative to the container.
   - `int verticalAlignment` — Vertical alignment of the text and icon relative to the container.
   - `int horizontalTextPosition` — Horizontal position of the text relative to the icon.
   - `int verticalTextPosition` — Vertical position of the text relative to the icon.

You specify these constraints by using the following final static variables:

```
TOP
BOTTOM
RIGHT
LEFT
CENTER
```

4 Call either of these versions of the `add` method on the command-aware container, depending on whether you want to add all command keys or a single command key:

```
add(Command cmd,
    RepresentationConstraints constraints)
```

```
add(Command cmd,
    String cmdKey,
    RepresentationConstraints constraints)
```

The `cmd` and `cmdKey` arguments are the same as described in “Adding Individual Command Keys” on page 126.

Example

Adding Commands with Representation Constraints to a Menu

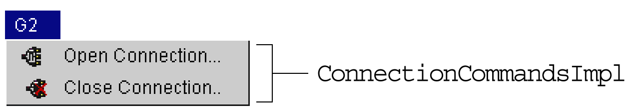
The following method creates a G2 menu that consists of a command represented as both text and an icon. In addition to the tasks that any command-aware container would perform, the method performs these tasks:

- Creates an instance of a `RepresentationConstraints` object, which uses both the textual and iconic descriptions of the command.
- Left-aligns the text and icon horizontally and centers the text and icon vertically within the menu.
- Positions the text to the right of the icon, and centers the text vertically relative to the icon.
- Adds the command to the menu by calling the `add` method that takes a command and a representation constraint object as arguments.

Here is the method that creates a G2 menu with representation constraints, where `this` refers to the application:

```
private CMenu createG2Menu() {
    CMenu menu = new CMenu("G2");
    connectionHandler = new ConnectionCommandsImpl(this);
    RepresentationConstraints constraints =
        new RepresentationConstraints
            (RepresentationConstraints.TEXT_AND_ICON,
             RepresentationConstraints.LEFT,
             RepresentationConstraints.CENTER,
             RepresentationConstraints.RIGHT,
             RepresentationConstraints.CENTER);
    menu.add(connectionHandler, constraints);
    return menu;
}
```

The menu looks like this:



Adding Separators

Often a menu or toolbar consists of logical groupings of commands. To help users recognize these logical groupings, you can add separators to the command-aware container. A **separator** is a horizontal bar in a menu and a vertical gap in a toolbar.

Alternatively, you can create a structured command, which automatically adds separators between command groups. For more information, see “Creating Commands with a Structure” on page 144.

To add a separator to a menu or toolbar:

➔ Call this method on the command-aware container in the location where you want the separator to appear:

```
addSeparator()
```

Example

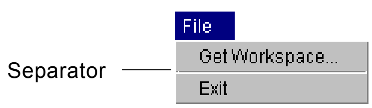
Adding a Separator to a Menu

This method creates a File menu with two commands, with a horizontal separator between the commands.

Here is the method that creates the File menu with separators:

```
private java.awt.Frame frame;
private com.gensym.ntw.TwGateway connection;

private CMenu createFileMenu() {
   CMenu menu = new CMenu ("File");
   exitHandler = new ExitCommandImpl(frame, connection);
   wkspHandler = new WorkspaceCommandsImpl(frame, connection);
   menu.add(wkspHandler,
            WorkspaceCommandsImpl.GET_KBWORKSPACE);
   menu.addSeparator();
   menu.add(exitHandler);
   return menu;
}
```

The menu looks like this:

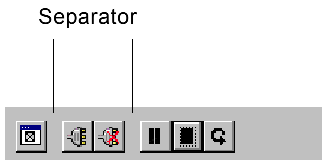


Adding Separators to a Toolbar

This method creates a toolbar with three commands and two separators, where the handlers are commands. The separators appear between each command.

```
private ToolBar createToolBar() {
   ToolBar tb = new ToolBar();
   tb.add(wkspHandler, WorkspaceCommandsImpl.GET_KBWORKSPACE);
   tb.addSeparator();
   tb.add(connectionHandler);
   tb.addSeparator();
   tb.add(g2StateHandler);
   return tb;
}
```

The toolbar looks like this:



Creating Commands

The `AbstractCommand` class provides a default implementation of the `Command` interface, which implements the command's behavior and handles event notification. `AbstractCommand` also implements `KeyableCommand`, which provides access to mnemonics and shortcuts for the command.

You create a command, using `AbstractCommand` by:

- Defining the command class.
- Implementing the constructor by calling the constructor for the superior class, providing an array of `CommandInformation` objects for each command action.
- Defining the action of the command by implementing its `actionPerformed` method.
- Delivering command events to command-aware containers by setting command properties.
- Getting command properties, as needed.
- Localizing the textual descriptions of the command.

The following sections describe these tasks in detail.

For a complete example of creating an abstract command, see "Example" on page 140.

Defining the Command Class

To define a command:

➔ Create a class that extends:

```
com.gensym.ui.AbstractCommand
```

For example, here is the general structure of the class definition for an abstract command that exits the application:

```
public class ExitCommandsImpl extends AbstractCommand {
    //Code goes here
}
```

Your command typically also implements some kind of listener so it can set command properties based on application events.

For more information, see "Delivering Command Events By Setting Properties" on page 135.

Implementing the Constructor

The constructor for an `AbstractCommand` subclass is responsible for initializing the command's properties. It typically takes as its argument one or more of the following:

- A connection, such as `com.gensym.ntw.TwAccess` or a `com.gensym.shell.util.ConnectionManager`.
- An application frame, such as a `com.gensym.mdi.MDIFrame` or a `java.awt.Frame`.
- An application, such as a `com.gensym.ntw.util.TW2Application` or `TW2MDIApplication`.

The `AbstractCommand` constructor takes an array of instances of the following class, one for each unique action in the command:

```
com.gensym.ui.CommandInformation
```

The constructor for a `CommandInformation` object takes these arguments in this order:

- `String key` — The command key, which the resources use as their lookup key to support localization.
- `boolean initialAvailability` — The command key's initial availability.
- `String shortResourceName` — The short resource file, which the command key uses to localize its textual description, or `null`.
- `String longResourceName` — The long resource file, which the command key uses to localize its textual description, or `null`.
- `String smallImageName` — The name of an image file for representing the command as a small icon.
- `String largeImageName` — The name of an image file for representing the command as a large icon.
- `boolean initialState` — The initial state of the command, which indicates whether the command is selected or unselected.
- `boolean immediate` — Whether or not the command key is executed immediately, where a value of `false` causes the command text to include ellipses (. . .).

- `String mnemonicResourceName` — The resource file that the command uses for translating its mnemonics.
- `KeyStroke shortcut` — The key sequence that the command uses as an accelerator.

If the specified resource file or image file is not an absolute path, the `AbstractCommand` looks for the file in the same directory as the command's class file.

Tip A `CommandInformation` object has another constructor, which allows you to specify explicitly the short and long descriptions, the small and large icons, and the mnemonic. You use this constructor when you do not know the description or icon to display until run time, or if you do not need to translate the mnemonic. If you use this constructor, you would not specify a resource for the corresponding description, icon, or mnemonic. See the API documentation for details.

To implement the constructor for an `AbstractCommand` subclass:

1 For each command key that the `AbstractCommand` subclass defines in its constructor, declare a final static variable as a `java.lang.String` and set it equal to a lookup key.

   The command uses this string as the lookup key into the resource properties files.

2 In the constructor for the `AbstractCommand` subclass, call the constructor for its superior class, passing in as the argument an array of `CommandInformation` objects for each command key.

In the following example, `CONNECT` and `DISCONNECT` are the command keys, and `OpenConnection` and `CloseConnection` are the lookup keys:

```
public static final String CONNECT = "OpenConnection";
public static final String DISCONNECT = "CloseConnection";
```

For information about defining the resource properties files and associated resources, see “Localizing Command Text and Mnemonics” on page 138.

This code fragment would appear in the constructor for a subclass of `AbstractCommand`. It supports two command keys, a short and long resource file for each key, two small toolbar button icons, a mnemonic, and a shortcut for each key. The shortcut for the `CONNECT` key is Ctrl + o, and the shortcut for the

DISCONNECT key is Ctrl + d. Call the getKeystroke static method on javax.KeyStroke to create the keystroke, and call Event.CTRL_MASK to pass in the Ctrl key as the accelerator.

```
super (new CommandInformation[]{
   new CommandInformation(CONNECT, true,
                          shortResource, longResource,
                          "connect.gif", null,
                          null, null
                          mnemonicResource,
                          KeyStroke.getKeyStroke('O',
                             Event.CTRL_MASK)),
   new CommandInformation(DISCONNECT, true,
                          shortResource, longResource,
                          "disconnect.gif", null,
                          null, null,
                          mnemonicResource,
                          KeyStroke.getKeyStroke('D',
                             Event.CTRL_MASK))})
```

Defining the Action of the Command

An AbstractCommand subclass must implement the actionPerformed abstract method to define the action of each command key. Here is the basic signature of this method:

```
public void actionPerformed(ActionEvent e) {
   //Action
}
```

You get the command key from the ActionEvent.

Once you have the command key for a particular ActionEvent, the actionPerformed method can test to see which command key the user executed, then provide the appropriate action for the particular key.

The implementation of the actionPerformed method typically tests to see whether the command key that gets returned is the correct key.

To define the action of each command key in the command:

1. Get the command key from the ActionEvent argument to the actionPerformed method of the command, by calling this method on the event, which returns a string:

   ```
   getActionCommand()
   ```

2. Implement the actionPerformed method, specifying the action for each command key.

For example, the following implementation of the `actionPerformed` method:

- Gets the command key from the `ActionEvent` argument.
- Throws an exception if the `ActionEvent` argument does not equal either of the command keys.
- Calls private methods to handle each action based on the command key.

Here is the implementation of the `actionPerformed` method for a command with two command keys:

```
public void actionPerformed(ActionEvent e) {
    String cmdKey = e.getActionCommand();
    if (!(cmdKey.equals (OPEN_CONNECTION)) &&
        !(cmdKey.equals (CLOSE_CONNECTION)))
          throw new IllegalArgumentException
              ("Unknown Key - " + cmdKey);
    if (cmdKey.equals(CONNECT))
       handleConnectCommand();
    if (cmdKey.equals(DISCONNECT))
       handleDisconnectCommand();
}
```

Delivering Command Events By Setting Properties

An `AbstractCommand` subclass typically implements some kind of listener so it can set one or more of its properties when an application event occurs. For example, a command might listen for changes in the connection status to G2, then set its availability to `true` when a connection opens and `false` when a connection closes.

When a command sets one of its properties, it notifies listeners of this command event. Because all command-aware containers implement the `CommandListener` interface, they automatically receive notification whenever a command property changes; thus, they automatically update their representation of the command in the container.

To set command properties and deliver command events:

➔ Call one of the following `set` methods on a subclass of `AbstractCommand` to notify listeners of the following events:

Call this method...	To notify listeners of this event...	Which determines...
`setAvailable`	`AVAILABILITY_CHANGED`	Whether the command is available or grayed out.
`setDescription`	`DESCRIPTION_CHANGED`	The textual description, which the command uses a menu text and tool tips.
`setState`	`STATE_CHANGED`	Whether the command is active or inactive.
`setIcon`	`ICON_CHANGED`	The iconic description.

The `set` methods all take as their first argument a command key, which determines the key whose property should be set. The methods also take whatever other arguments are appropriate, such as a `boolean`, a `String`, or an icon.

The event types are final static variables defined on `CommandEvent`, whose values are integers.

Examples

Setting the Initial Availability in the Constructor

You specify the initial availability of an abstract command in its constructor. For example, this code fragment sets the initial availability of the `CONNECT` and `DISCONNECT` command keys of a command to `false` if the current connection does not exist.

```
private com.gensym.ntw.TwGateway connection;

if (connection == null){
   setAvailable(CONNECT, false);
   setAvailable(DISCONNECT, false);
}
```

Setting the Availability When an Event Occurs

Your command might implement the `com.gensym.shell.util.ContextChangedListener` to receive notification when the current connection context changes.

The following listener method makes the `CONNECT` key available and the `DISCONNECT` key unavailable when the current connection context changes:

```
public void currentConnectionChanged(ContextChangedEvent e) {
   TwAccess context = e.getConnection();
   if (context == null)
      setAvailable(CONNECT, true)
   else
      setAvailable(DISCONNECT, false)
}
```

Getting Command Properties

An `AbstractCommand` subclass might need to get the current value of one of its properties, such as its state or availability. For example, when you define the action of the command through its `actionPerformed` method, you might need to test whether the command key is available before performing its action.

To provide another example, the state of one command key might depend on the state of another command key, such that selecting one command key causes the other command key to become unselected.

To get command properties:

➔ Call one of the following methods on a subclass of `AbstractCommand`:

Call this method...	To determine...
`isAvailable`	Whether the command is available or unavailable, as a `boolean`.
`getDescription`	The textual description.
`getState`	Whether the command is active or inactive, as a `boolean`.
`getStructuredKeys`	The structure of an abstract structured command.
`getIcon`	The iconic description.

Call this method...	To determine...
`getMnemonic`	The mnemonic as a `java.lang.Character`.
`getShortcut`	The shortcut as a `javax.swing.KeyStroke`

Localizing Command Text and Mnemonics

The `AbstractCommand` class has built-in support for localizing textual descriptions and mnemonics by providing these arguments in the `CommandInformation` constructor:

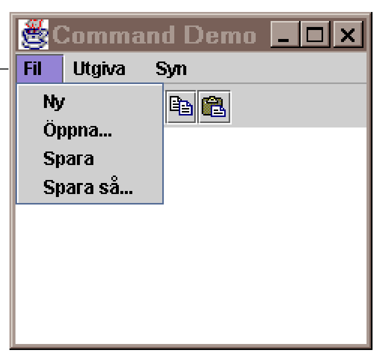
- `String key`
- `String shortResourceName`
- `String longResourceName`
- String mnemonicResourceName

If the `CommandInformation` object does not provide the short and long description, or the mnemonic explicitly in its constructor, the `AbstractCommand` subclass uses the command key as the lookup key into the resource files.

The `AbstractCommand` subclass performs the localization once per key.

For example, this figure shows an application whose menu labels are localized for the Swedish language:

Localized menu and menu choice text.



The basic steps for localizing textual descriptions and mnemonics for commands are:

- Create a short resource properties file that provides localized text strings for each lookup key, which menus use as labels.
- Create a long resource properties file that provides additional localized text strings for each lookup key, which a `ToolBar` uses as a tool tip.

- Create a mnemonic resource properties file that provides localized characters for each lookup key, which menus use as mnemonics.
- Create a resource bundle.

The following sections provide examples of each of these steps.

For general information on localization, see Appendix A, “Localization.”

Examples

Creating a Short Resource Properties File

To localize the textual representation of a command, create a short resource properties file that contains pairs of keys and short descriptions for each command key. The command uses the short description as its textual representation.

For example, you might create a short resource properties file named `CommandShortResources.properties` in the same directory as the source code for the `AbstractCommand` subclass. This file might contain the following keys and short descriptions for the `CONNECT` and `DISCONNECT` command keys:

```
OpenConnection=Open Connection
CloseConnection=Close Connection
```

To cause the textual description to contain ellipses (`...`) to indicate that the command displays a dialog, specify the `immediate` argument to the `CommandInformation` object as `false`.

Creating a Long Resource Properties File

To support tool tips for iconic descriptions of a command, create a long resource properties file that contains pairs of keys and long descriptions for each command key. A `ToolBar` use these long descriptions as a tool tip when the cursor lingers over the iconic representation of each command.

For example, you might create a long resource properties file named `CommandLongResources.properties` in the same directory as the source code for the `AbstractCommand` subclass. This file might contain the following keys and long descriptions for the `CONNECT` and `DISCONNECT` command keys:

```
OpenConnection=Opens a new connection to G2 on a host and port
CloseConnection=Closes the selected G2 connection
```

Creating a Mnemonic Resource Properties File

To support mnemonics for commands, create a mnemonic resource properties file that contains pairs of keys and a single alpha-numeric character that is the mnemonic. A menu underlines the first occurrence of the mnemonic in the short description.

To execute the mnemonic, enter Alt, followed by the menu mnemonic, followed by the command mnemonic. For example, to execute the mnemonic for the G2 > Open Connection command, you might enter Alt + g + o.

For example, here is a mnemonic resource file for the `CONNECT` and `DISCONNECT` command keys:

```
OpenConnection=O
CloseConnection=C
```

To create the mnemonic for the top-level menu, call `setMnemonic`, a method on `javax.swing.JMenu`. For an example, see the `com.gensym.shell.Shell` class.

Creating a Resource

Create resources for the short and long properties files by calling the `getBundle` static method on a `Resource`, providing a string as its argument, which names the resource properties file. For example:

```
private com.gensym.message.Resource shortBundle =
    Resource.getBundle("com.gensym.demos.test.CommandShortResources");

private com.gensym.message.Resource longBundle =
    Resource.getBundle("com.gensym.demos.test.CommandLongResources");
```

If the resource is a fully qualified class name, the command looks for the resource in the same directory as the command class.

Example

This section provides a complete example of creating a command that exits the application and closes any open connections, if they exist. The command has these features:

- Defines a single command key, which exits the application.
- Listens for window events and implements specific behavior for the window closing event.
- Closes any open connections before exiting.
- Localizes menu text, tool tips, and mnemonics by providing short and long resource properties files and a mnemonics resource properties file.
- Supports textual and iconic representations.
- Supports a mnemonic and shortcut (Alt + z).

The following sections provide explanations of the code.

```
import java.awt.Frame;
import java.awt.event.WindowListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.ActionEvent;
import com.gensym.core.ExitThread;
import com.gensym.ntw.TwAccess;
import com.gensym.message.Resource;
import com.gensym.message.Trace;
import com.gensym.shell.util.ConnectionManager;
import com.gensym.ui.AbstractCommand;
import com.gensym.ui.CommandInformation;

//Defining the command
public final class ExitCommand extends AbstractCommand {

    //Private variables
    public static final String EXIT = "Exit";
    private static final String shortResource = "CommandShortResource";
    private static final String longResource = "CommandLongResource";
    private static final String mnemonicResource =
        "MnemonicResource";
    private Resource i18n = Resource.getBundle
        ("com.gensym.shell.commands.Errors");
    private ConnectionManager connectionMgr = null;
    private TwAccess singleConnection = null;
    private WindowListener windowClosingAdapter;

    //Constructor for single connection application
    public ExitCommand(Frame frame, TwAccess connection) {
        this(frame, (ConnectionManager)null);
        singleConnection = connection;
        //Handle window closing event
        windowClosingAdapter = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                if (singleConnection != null)
                    exitApp(singleConnection);
                else if (connectionMgr != null)
                    exitApp(connectionMgr);
                else
                    exitApp();
            }
        };
        frame.addWindowListener(windowClosingAdapter);
    }
```

```
    //Constructor for multiple connection application
    public ExitCommand(Frame frame,
                        ConnectionManager connectionManager) {
        super(new CommandInformation[]{
            new CommandInformation(EXIT, true,
                                shortResource, longResource,
                                "exit_tw.gif", null, null, true,
                                mnemonicResource,
                                KeyStroke.getKeyStroke('Z',
                                    Event.ALT_MASK))});
        connectionMgr = connectionManager;
    }

    //Provide method for setting the connection
    public void setConnection(TwAccess connection){
        singleConnection = connection;
    }

    //Implement ActionListener interface method
    public void actionPerformed(ActionEvent e){
        String cmdKey = e.getActionCommand();
        if (cmdKey == null) return;
        if (!isAvailable(cmdKey)){
            throw new IllegalStateException
                (i18n.format("CommandIsUnavailable", cmdKey));
        }
        if (singleConnection != null)
            exitApp(singleConnection);
        else if (connectionMgr != null)
            exitApp(connectionMgr);
        else exitApp();
    }

    //Close connections and exit in multiple connection applications
    private void exitApp(ConnectionManager connectionManager) {
        TwAccess[] cxns = connectionManager.getOpenConnections();
        for (int i=0; i<cxns.length; i++)
            cxns[i].closeConnection();
        exitApp();
    }

    //Close connections and exit in single connection applications
    private void exitApp(TwAccess connection) {
        connection.closeConnection();
        exitApp();
    }

    //Close connections and exit when no connection exists
    private void exitApp() {
        System.exit(0);
    }
}
```

Defining the Command

The `ExitCommand` class:

- Extends `AbstractCommand` so it can automatically deliver command events when the state, availability, description, or icon changes, and so it can support internationalization and iconic representations.
- Implements `java.awt.event.WindowListener` so it can listen for window closing events.

The string `"Exit"` is the command key into the `CommandShortResource.properties`, `CommandLongResource.properties`, and `MnemonicResource.properties` files, which support localization of textual descriptions and mnemonics. These properties files are located in the same package as the command class.

Creating the Constructor for a Single Connection Application

For a single connection application, the `ExitCommand` class needs to know about the application frame and the G2 connection. This constructor calls the constructor for a multiple connection application, passing `null` as the connection.

The constructor for a single connection application initializes a variable for the connection, creates a `WindowAdapter` for handling window closing events, as the next section describes, and adds the command as a `WindowListener`.

Handling Window Closing Event

The `ExitCommand` class implements the `java.awt.event.WindowListener` interface, which means it is notified of standard windows events, such as when the window is closing.

The constructor creates a `WindowAdapter` as an inner class and implements the `windowClosing` method, which has the same implementation as the `actionPerformed` method.

The command for exiting the application is always available; otherwise, this method would also set its availability.

Creating the Constructor for a Multiple Connection Application

For a multiple connection application, the `ExitCommand` class needs to know about the application frame and the `ConnectionManager`. The constructor calls the constructor on its superior class to create the command key for the command, and it initializes a variable for the `ConnectionManager`.

Providing a Method for Setting the Connection

The `setConnection` method provides a method for setting the connection in a single connection application.

Implementing the ActionListener Interface Method

If the command key associated with the action event is not `EXIT`, the command throws an exception, indicating that the argument is an illegal type.

The `actionPerformed` method calls one of three versions of the `exitApp` method, depending on the type of connection and whether the connection exists.

Closing All Connections and Exiting

When the user invokes the action of the command, the command calls different versions of the `exitApp` method, depending on whether a connection currently exists, and if so, whether the command was created with a single connection or a `ConnectionManager`.

If a connection exists, the `exitApp` method:

- Closes the current connection.
- Exits the application.

If no current connection exists, the method simply exits the application.

Creating Commands with a Structure

The `AbstractStructuredCommand` class provides a default implementation of this interface:

```
com.gensym.ui.StructuredCommand
```

`AbstractStructuredCommand` handles command behavior and event notification, and provides a set of related actions with one or more of the following features:

- A hierarchical structure.
- A logical grouping.
- Dynamically updating structure.

Command-aware containers represent `AbstractStructuredCommands` differently, depending on the type of container. For example, this table describes the results of adding an `AbstractStructuredCommand` subclass with different features to a `CMenu`:

If you add a command with this feature...	The menu...
Hierarchical structure	Represents the command as a submenu.
Command group	Includes a separator between the groups.

Subclasses of `AbstractStructuredCommand` notify listeners when the structure of the command changes, which means command-aware containers automatically update the representation of the command.

Otherwise, the process of adding an `AbstractStructuredCommand` subclass to a command-aware container is identical to that of adding an `AbstractCommand` subclass, as described in "Creating Command-Aware Containers" on page 122.

The process for creating a subclass of `AbstractStructuredCommand` is identical to that of creating a subclass of `AbstractCommand`, with these key differences:

To...	Do this...
Define the command class	Extend `AbstractStructuredCommand`.
Implement the constructor	Call the constructor for the superior class by providing an array of `StructuredCommandInformation` objects for each command key in the structure. Each key can represent a subcommand, a command group, or an individual command.
Deliver structured command events	Set a property of the structured command.
Get the structure	Get the structure itself or individual command elements of the structure.

In all other ways, creating an `AbstractStructuredCommand` subclass is identical to creating an `AbstractCommand` subclass, described in "Creating Commands" on page 131.

The following headings describe the differences in creating a command with a structure compared with creating a simple command.

Defining the Command Class

To define a command with a structure:

➔ Create a class that extends this class:

```
com.gensym.ui.AbstractStructuredCommand
```

Here is the general structure of the class definition for a command that switches the current connection:

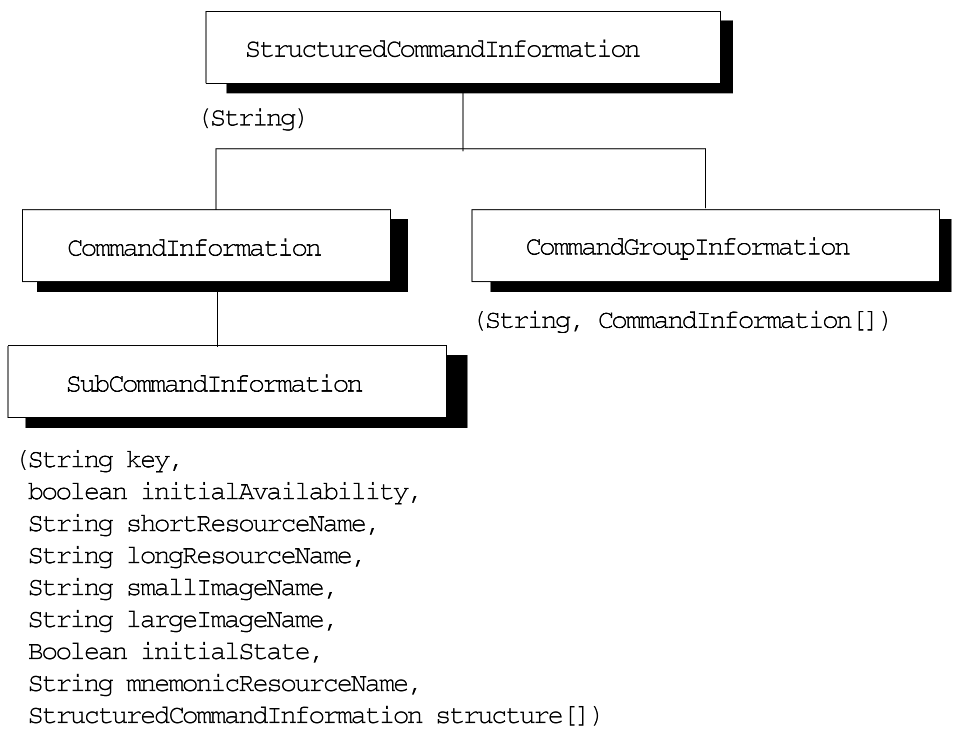
```
public class SwitchConnectionCommandImpl
   extends AbstractStructuredCommand {
   //Command code
}
```

Implementing the Constructor

The constructor for an `AbstractStructuredCommand` subclass takes an array of instances of this class, which initializes the structure:

```
com.gensym.ui.StructuredCommandInformation
```

`StructuredCommandInformation` is the superior class of a hierarchy of objects, all of which are in the `com.gensym.ui` package. In the following figure, the arguments appear below each type of object:



Tip Similar to `CommandInformation`, `SubCommandInformation` has another constructor, which allows you to specify explicitly the short and long descriptions, the small and large icons, and the mnemonic. See the API documentation for details.

For information on the arguments to `CommandInformation`, see “Implementing the Constructor” on page 132.

You use instances of each of these classes to create different command structures, as this table describes:

Use this type of object...	To create this command structure...
`SubCommandInformation`	A top-level action with associated subactions, which can include one or more instances of this class: `StructuredCommandInformationObject`
`CommandGroupInformation`	A group of related actions separated from other groups with a separator, which can include one or more instances of this class: `CommandInformationObject`
`CommandInformation`	A single action.

Each command-aware container represents the structured command and separators appropriately for the type of container in which it appears.

To call the constructor for an `AbstractStructuredCommand` subclass:

➔ In the constructor for the `AbstractStructuredCommand` subclass, call the constructor for its superior class, passing in as the argument an array of `StructuredCommandInformation` objects that describe the command structure.

The first example that follows creates a structured command with two command groups:

- Cut, Copy, and Paste
- Delete

The following examples show the result of adding the structured command to various command-aware containers.

Examples

Creating a Subcommand with Two Command Groups

The following example defines an `AbstractStructuredCommand` subclass called `EditCommands`. The command consists of a single `SubCommandInformation` object, whose key is `EDIT`. The `SubCommandInformation`, in turn, consists of two `CommandGroupInformation` objects, whose keys are:

- `CutCopyPaste`, which consist of three individual `CommandInformation` objects, whose keys are:

  ```
  CUT
  COPY
  PASTE
  ```

- `Delete`, which consists of a single `CommandInformation` object, whose key is:

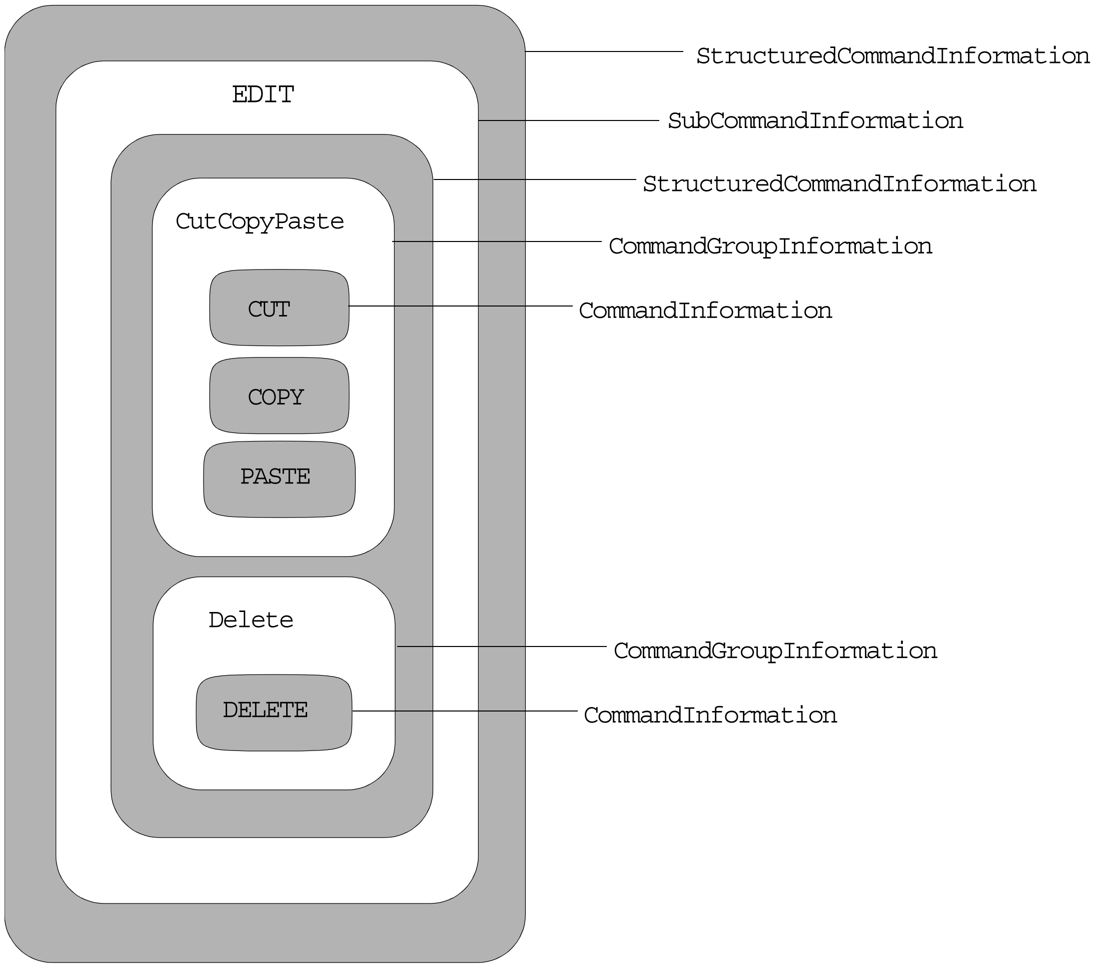
  ```
  DELETE
  ```

Here is a static method that you can call in the constructor for `EditCommands` to create the structured command:

```
private static StructuredCommandInformation[] buildCommandStructure() {

    //Build the "cut/copy/paste" group
    CommandInformation
        cut = new CommandInformation(CUT, true,
                                     shortResource,
                                     longResource,
                                     "cut.gif", null,
                                     null, true, null, null),
        copy = new CommandInformation(COPY, true,
                                      shortResource,
                                      longResource,
                                      "copy.gif", null,
                                      null, true, null, null),
        paste = new CommandInformation(PASTE, true,
                                       shortResource,
                                       longResource,
                                       "paste.gif", null,
                                       null, true, null, null);
    CommandGroupInformation
        cutCopyPasteGroupInfo =
            new CommandGroupInformation("CutCopyPaste",
                                        new CommandInformation[]
                                        {cut, copy, paste});
```

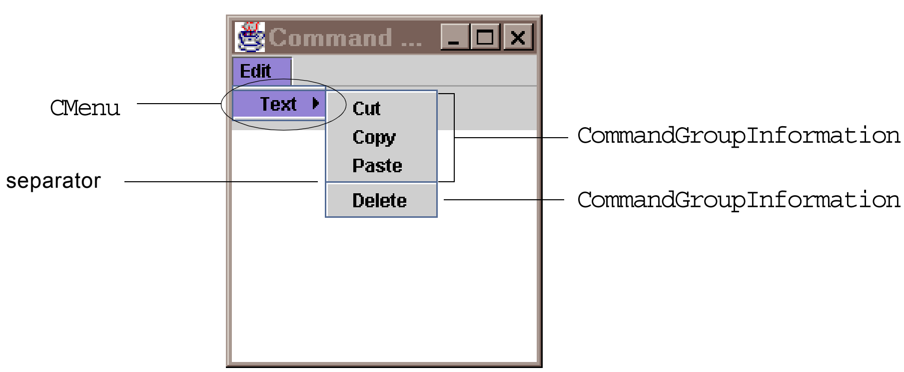
```
    //Build the "delete" group
    CommandInformation
        delete = new CommandInformation(DELETE, true,
                                        shortResource,
                                        longResource,
                                        null, null,
                                        null, true, null, null);
    CommandGroupInformation
        deleteGroupInfo =
            new CommandGroupInformation("Delete",
                                        new CommandInformation[]
                                        {delete});

    SubCommandInformation subCommandInfo =
        new SubCommandInformation (EDIT, true,
                                   shortResource,
                                   longResource,
                                   null, null,
                                   null, null,
                                   new StructuredCommandInformation[]
                                   {cutCopyPasteGroupInfo,
                                    deleteGroupInfo});

    return new StructuredCommandInformation[] {subCommandInfo};
}

//Constructor
public EditCommands() {
    super(buildCommandStructure());
}
```

Here is a conceptual representation of the structured command:



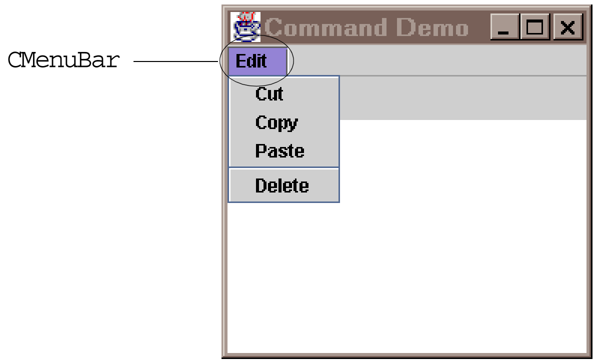
Adding a Structured Command to a CMenu

Here is how the structured command looks when added to a CMenu:



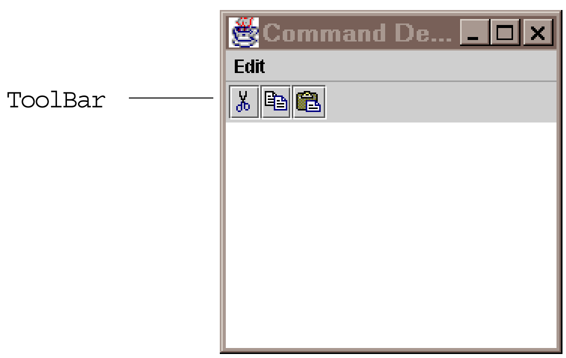
Adding a Structured Command to a CMenuBar

Here is how this command looks when added to a CMenuBar:



Adding a Structured Command Key to a ToolBar

Here is how this command looks when adding the CutCopyPaste command group to a ToolBar:



Delivering Structured Command Events by Setting Properties

In addition to the properties you can set for any subclass of AbstractCommand, you can set the structure of a subclass of AbstractStructuredCommand.

Setting the structure notifies registered StructuredCommandListeners of the event by delivering an instance of this event class:

```
com.gensym.ui.CommandEvent
```

To set the structure of a structured command and deliver the associated event:

➔ Call this method on a subclass of AbstractStructuredCommand:

```
setStructuredKeys(StructuredCommandInformation structure[])
```

Calling this method notifies listeners of the following event, which is a static final variable defined on `CommandEvent`, which is an integer:

```
STRUCTURE_CHANGED
```

Example

Creating a Subcommand that Updates Dynamically

You might want to create a structured command whose contents update dynamically, based on the state of the application. To do this, you typically create a subcommand whose contents is initially empty, then update the command structure when an application event occurs.

The following example creates a structured command that switches the current connection dynamically.

Note The command is only relevant in the context of an application that supports multiple connections to G2 through a `com.gensym.shell.util.ConnectionManager`.

The command creates its structure by using a `SubCommandInformation` object, whose contents is initially empty.

The command listens for changes in the connection status by using two adapter classes. Each adapter class implements a single method, which updates the command's structure when the listener event occurs. This table describes the adapter classes, the classes they extend, and the methods they implement:

This adapter class...	Implements this interface...	And defines this abstract method...
`ContextChangedAdapter`	`ContextChangedListener`	`currentConnectionChanged`
`G2ConnectionAdapter`	`G2ConnectionListener`	`g2ConnectionClosed`

When each of the corresponding events occurs, the abstract method calls the `setStructuredKeys` method on the `AbstractStructuredCommand` subclass, which updates the list of available connections in the subcommand, as this table describes:

When this event occurs...	The structured command...
Current connection changes	Adds the connection to the subcommand, selects it as the current command, and makes the subcommand available.
Current connection closes	Removes the connection from the subcommand and makes the subcommand unavailable if no connection exists.

Here is the structured command whose contents update dynamically:

```
import java.util.Vector;
import java.util.Hashtable;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import com.gensym.ntw.TwAccess;
import com.gensym.jgi.G2ConnectionListener;
import com.gensym.jgi.G2ConnectionAdapter;
import com.gensym.jgi.G2ConnectionEvent;
import com.gensym.jgi.ConnectionTimedOutException;
import com.gensym.jgi.G2AccessInitiationException;
import com.gensym.jgi.G2AccessException;
import com.gensym.ntw.TwGateway;
import com.gensym.message.Resource;
import com.gensym.shell.util.*;
import com.gensym.ui.AbstractStructuredCommand;
import com.gensym.ui.CommandInformation;
import com.gensym.ui.SubCommandInformation;
import com.gensym.ui.StructuredCommandInformation;

//Abstract structured command definition
public final class ChangeConnectionCommand
   extends AbstractStructuredCommand {

   //Public varible
   public static final String TW_SWITCH_CONNECTION =
      "TwSwitchConnection";

   //Private variables
   private static final String shortResource =
      "CommandShortResource";
   private static final String longResource = "CommandLongResource";
   private Resource i18n = Resource.getBundle("Errors");
```

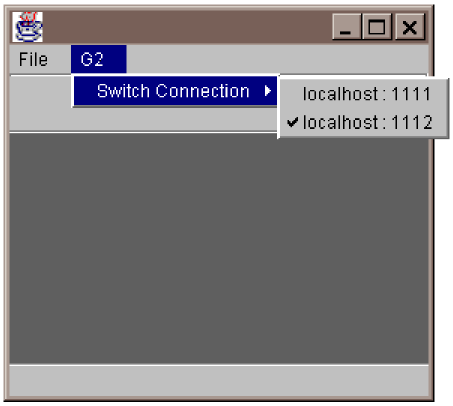
```
    private Resource shortBundle =
       Resource.getBundle("CommandShortResource");
    private Hashtable connectionTable;
    private ConnectionManager connectionMgr;
    private G2ConnectionAdapter closingListener;
    private TwAccess previousConnection;
    private TwAccess currentConnection;
    private Vector connectionList;
    private ContextChangedListener contextChangedListener;

    //Contructor
    public ChangeConnectionCommand(ConnectionManager connectionMgr){

       //Create empty array of command information objects
       super (new CommandInformation[]{});

       //Initialize properties
       this.connectionMgr = connectionMgr;
       currentConnection = connectionMgr.getCurrentConnection();
       connectionTable = new Hashtable();
       contextChangedListener = new ContextChangedAdapter();
       connectionList = new Vector();

       //Add listeners to abstract structured command
       connectionMgr.addContextChangedListener(contextChangedListener);
       closingListener = new G2CloseAdapter();

       //Get open connections and names, create hash table,
       //and add each as a closing listener
       TwAccess[] openConnections = connectionMgr.getOpenConnections();
       for (int i=0; i<openConnections.length; i++){
          String connectionName = openConnections[i].toShortString();
          connectionTable.put(connectionName, openConnections[i]);
          openConnections[i].addG2ConnectionListener(closingListener);
        connectionList.addElement(connectionName);
    }

    //Define command structure
    setStructuredKeys(new CommandInformation[]
       {createSwitchSubCommand()});

    //Set command availability
    if (connectionMgr.getCurrentConnection() == null){
       setAvailable(TW_SWITCH_CONNECTION, false);
    }
 }

 //Create SubCommandInformation object
 private SubCommandInformation createSwitchSubCommand(){

    //Create array of CommandInformation objects for connections
    CommandInformation[] connections = new
       CommandInformation[connectionList.size()];
```

```
    //Get each connection name from table and make available
    for (int i=0; i<connectionList.size(); i++){
        String connectionName = (String)connectionList.elementAt(i);
        TwAccess connection =
            (TwAccess)connectionTable.get(connectionName);
        Boolean state = Boolean.FALSE;
        if (connection.equals(currentConnection))
            state = Boolean.TRUE;

        //Create CommandInformation object for each connection
        connections[i] = new CommandInformation
            (connectionName, true, null, null, null, null, state,
             true, connectionName, connectionName, null, null);
    }

    //Make subcommand available if any connections exists
    boolean available = connectionList.size() > 0;

    //Return SubCommandInformation object with connections
    return new SubCommandInformation(TW_SWITCH_CONNECTION,
                                     available, shortResource,
                                     longResource, null, null,
                                     null, null, connections);
}

//Implement ContextChangedListener to add connections
//to the subcommand when the current connection changes
class ContextChangedAdapter implements ContextChangedListener{
    public void currentConnectionChanged(ContextChangedEvent e){
        TwAccess newCurrentConnection = e.getConnection();
        previousConnection = currentConnection;
        currentConnection = newCurrentConnection;
        if (previousConnection != null)
        setState(previousConnection.toShortString(), Boolean.FALSE);

        //If no current connection exists, make command unavailable
        if (currentConnection == null){
            setAvailable(TW_SWITCH_CONNECTION, false);
        }

        //Else, create new connection, add to table with name,
        //make command available, and add as a closing listener
        else{
            String connectionName = currentConnection.toShortString();
            if (connectionTable.get(connectionName) == null){
                setAvailable(TW_SWITCH_CONNECTION, true);
                connectionList.addElement(connectionName);
                connectionTable.put(connectionName, currentConnection);
                currentConnection.addG2ConnectionListener
                    (closingListener);
```

```
                //Set the structure for the command, adding
                //new connection to the CommandInformation objects
                setStructuredKeys new CommandInformation[]
                   {createSwitchSubCommand()});
             }
         //Make the new connection be the selected connection
         setState(connectionName, Boolean.TRUE);
         }
     }
}
//Implement G2ConnectionAdapter to remove connections
//from the subcommand when a connection closes
class G2CloseAdapter extends G2ConnectionAdapter{
    public void g2ConnectionClosed(G2ConnectionEvent e){
         TwAccess connection = (TwAccess)e.getSource();
         String connectionName = connection.toShortString();
         //Remove connection from list and table
         connectionList.removeElement(connectionName);
         connectionTable.remove(connectionName);
         //Update the command structure
         setStructuredKeys(new CommandInformation[]
             {createSwitchSubCommand()});
         //Make subcommand unavailable if no connections exist
         if (connectionList.size() == 0){
             setAvailable(TW_SWITCH_CONNECTION, false);
         }
     }
}
```

Adding a Dynamically Updating Subcommand to a Menu

This example shows the result of adding to a menu a dynamically updating structured command that displays the current connections:



Getting the Structure

You can get the structure of a structured command or get the individual `StructuredCommandInformation` objects from the command structure. The `CommandUtilities` class provides a convenience method that lets you get individual elements from the structure.

To get the structure:

➔ Call this method on a subclass of `AbstractStructuredCommand`:

```
getStructuredKeys()
```

This method returns an array of `StructuredCommandInformation` objects from the `AbstractStructuredCommand` subclass.

To get the element associated with a key in the structure:

1 Create an instance of this class:

```
com.gensym.ui.CommandUtilities
```

2 Call this method on `CommandUtilities`:

```
getElementForKey(StructuredCommand command, String key)
```

This method traverses the structure of any implementation of the `StructuredCommand` interface, and returns the string associated with the first command key or `SubCommand` key that equals the specified string.

Implementing the Command Interface

You implement the `Command` interface to create a command that:

- Handles notification of command events explicitly.
- Provides its own implementation of the `Command` interface methods.

Otherwise, it is simpler to extend one of the abstract command classes that provide default implementations of the `Command` interface, as described in:

- “Creating Commands” on page 131.
- “Creating Commands with a Structure” on page 144.

To create a command that handles its own events and implements its behavior:

1. Create a class that implements this interface:

   ```
   com.gensym.ui.Command
   ```

2. Implement the `actionPerformed` method to specify the behavior of the command.
3. Implement a listener to handle event notification explicitly, as needed.
4. Provide implementations for the abstract command methods to:
   - Get and set the command properties.
   - Add and remove command listeners.
   - Return the command text as a string.

Example

The following example shows an implementation of the `Command` interface that exits the application. This example provides a comparison with the example described in “Example” on page 140, which creates an `AbstractCommand` subclass for exiting the application.

The command implements the `java.beans.PropertyChangeListener` to listen for changes in any of the properties of the command.

The command provides implementations of these abstract methods:

- `isAvailable` and `isImmediate`, which set initial values of properties.
- `getKeys`, `getDescription`, `getIcon`, `getState`, which get properties of the command.
- `addCommandListener` and `removeCommandListener`, which add and remove command listeners.
- `toString`, which returns the command text as a string.

This command does not handle any command events, thus command-aware containers are not notified when a command event occurs.

For an example of handling event notification explicitly, see the source code for this class:

```
com.gensym.demos.singlecxnsdiapp.ConnectionCommandImpl
```

Here is the complete code for the `ExitCommandImpl`:

```
package com.gensym.demos.singlecxnsdiapp;

import com.gensym.ui.Command;
import com.gensym.ui.CommandListener;
import java.awt.Frame;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;
import com.gensym.message.Resource;
import com.gensym.ntw.TwAccess;

public class ExitCommandImpl implements Command {

    //Private variables
    private static final String EXIT = "exit";
    private static Resource i18nShort = Resource.getBundle
        ("com.gensym.demos.wksppanel.ShortCommandLabels");
    private static Resource i18nLong  = Resource.getBundle
        ("com.gensym.demos.wksppanel.LongCommandLabels");
    private static final Class thisClass = ExitCommandImpl.class;
    private TwAccess connection = null;

    //Constructor
    public ExitCommandImpl (Frame frame) {
        WindowListener windowClosingAdapter = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            exitApp();
        }
    };
    frame.addWindowListener(windowClosingAdapter);
    }

    //Method to set the current connection
    public void setConnection(TwAccess cxn){
        connection = cxn;
    }
```

```
    //Method for exiting and closing connection
    public void exitApp() {
        if (connection != null)
        connection.closeConnection();
        System.exit (0);
    }

    //Implement ActionListener interface methods
    public void actionPerformed(ActionEvent e) {
        exitApp();
    }

    //Implement Command interface methods
    public boolean isImmediate(String key){
        return true;
    }

    public boolean isAvailable (String cmdKey) {
        return true;
    }

    public String[] getKeys() {
        return new String[] {EXIT};
    }

    public String getDescription (String cmdKey, String key) {
        if (!cmdKey.equals (EXIT))
            throw new IllegalArgumentException
                ("Unsupported key - " + cmdKey);
        if (key.equals (Command.SHORT_DESCRIPTION))
            return i18nShort.getString (cmdKey);
        else
        return i18nLong.getString (cmdKey);
    }

    public Image getIcon (String cmdKey, String key) {
        if (!cmdKey.equals (EXIT))
        throw new IllegalArgumentException ("Unsupported key - " +
cmdKey);
     if (key.equals (Command.SMALL_ICON))
       return Toolkit.getDefaultToolkit().getImage(getClass().
            getResource (cmdKey + "small.gif"));
    else
        return Toolkit.getDefaultToolkit().
            getImage(getClass().getResource (cmdKey + "large.gif"));
    }

    public Boolean getState(String cmdKey) {
        return null;
    }
```

```
    public void addCommandListener (CommandListener listener) {
       // Do nothing.
    }

    public void removeCommandListener (CommandListener listener) {
       // Do nothing.
    }

    public String toString() {
       return "Exit Command";
    }
    //End Command interface methods
}
```

Overriding Mnemonics and Shortcuts for Shell Commands

The commands in the `com.gensym.shell.commands` package provide default mnemonics for all command keys, and default shortcuts for certain command keys. For a list of these defaults, see “Using Menu Command Mnemonics and Shortcuts” on page 58.

You can override the default mnemonics and shortcuts, or provide additional shortcuts by creating a `KeyInformation` object for individual command keys. You then pass `KeyInformation` objects to the `add` method on the command-aware container that is adding the command.

To override the mnemonics and shortcuts for a shell command:

1 Create a `com.gensym.ui.KeyInformation` using the following constructor:

```
(String cmdKey,
 boolean useDefaultMnemonic, Character mnemonic,
 boolean useDefaultShortcut, KeyStroke shortcut)
```

For example, to override just the mnemonic for the `RESET` key for an instance of `ConnectionCommands` to be “R”, create a `KeyInformation` as follows:

```
KeyInformation keyInfo =
   new KeyInformation(RESET, false, new Character('R'),
                      true, null)
```

To override just the shortcut, set `useDefaultShortcut` to `false` and provide a `javax.swing.KeyStroke`. To override both, set both `boolean` values to `false` and provide both a mnemonic and shortcut.

2 Pass in an array of `KeyInformation` objects to the `add` method of a command-aware container that is adding the command, using this method:

```
add(Command cmd, String cmdKey,
    RepresentationConstraints constraints,
    KeyInformation[] mnemonicAndShortcutOverrides)
```

For example, this code fragment overrides the mnemonic for the `RESET` command key when adding it to a `CMenu`:

```
private com.gensym.message.Resource bundle =
   Resource.getBundle("com.gensym.shell.Messages")

private CMenu createG2Menu() {
   CMenu menu = new CMenu(bundle.getString("G2Menu"));
   connectionHandler = new ConnectionCommandsImpl(this);
   menu.add(connectionHandler, ConnectionCommandsImpl.CONNECT,
            null, new KeyInformation[] {keyInfo});
   return menu;
}
```

Creating Palettes

Describes how to create palettes from commands.

Introduction

A **palette** is a container that you use to create objects and place them in some other container. For example, your application might include a palette of G2 objects, which you place in a workspace view to create a schematic diagram.

A palette consists of **palette buttons**, which are icons that represent the object you want to place. To place the object in a container, the end user double-clicks the button, then clicks anywhere in the container.

The `com.gensym.wksp.ScalableWorkspaceView` component responds appropriately when the user clicks a palette button, then clicks in a workspace view by:

- Changing the cursor to indicate that a palette button has been clicked.
- Representing the item in the workspace view by using the icon description of the palette button.
- Creating an item of the appropriate type when the user clicks a location in the workspace view after double-clicking a palette button.

These packages provide classes that let you create generic palettes of objects and palettes of G2 object:

```
com.gensym.ui.palette

com.gensym.ui

com.gensym.ntw.util
```

Palettes and Palette Buttons

You create a palette in one of two ways:

- By adding instances of an implementation of the `ObjectCreator` interface to a `Palette`, using versions of the `add` method, where each key associated with the `ObjectCreator` generates a `PaletteButton`.
- By explicitly adding instances of a `PaletteButton` to a `Palette`.

A `PaletteButton` represents a single item that you can create from a `Palette`. The type of item that gets created depends on how you create the `PaletteButton`, as follows:

If you create palette buttons by...	Then...
Adding instances of an implementation of `ObjectCreator` to a `Palette`	The `ObjectCreator` determines the type of item the `PaletteButton` creates.
Adding instances of a `PaletteButton` directly to the `Palette`	You specify the object to create directly in the `PaletteButton`, or you create and set the `ObjectFactory`, which determines the type of item to create.

You can add all `ObjectCreator` keys or individual keys to the `Palette`, with or without `RepresentationConstraints`. You can add separators to the palette.

A `Palette` responds appropriately to changes in an `ObjectCreator` through the `ObjectCreatorListener` interface.

For information about creating palettes, see “Creating a Palette of Objects” on page 169.

Object Creators

An ObjectCreator is an interface for creating instances of classes. Implementations of the ObjectCreator interface provide a unique key to represent each item, which is typically a unique class. The ObjectCreator uses an ObjectFactory to determine the object to create for each key.

An ObjectCreator defines the following properties for the item it represents:

- Availability — Whether the item is available for double-clicking or whether it is grayed out.
- Description — The textual description that represents the item in the palette.
- Icon — The icon that represents the item in the palette.

A Palette uses the ObjectCreator's short description as the button's textual representation if you add the ObjectCreator with constraints. A Palette uses the ObjectCreator's long description when the cursor lingers over the palette button.

An ObjectCreator delivers an ObjectEvent whenever the availability, description, or icon of an ObjectCreator key changes. Clients can add themselves as ObjectCreatorListeners to receive notification when the availability, description, or icon of an individual key changes.

For information about using ObjectCreators, see “Creating Buttons from an ObjectCreator” on page 170.

Structured Object Creators

A StructuredObjectCreator is an interface that defines an ObjectCreator with one or more of the following features:

- A logical groupings of items.
- A hierarchical structure of items.
- A dynamically created group of items.

You can add all StructuredObjectCreator keys or individual keys to the Palette, with or without RepresentationConstraints. Each group in the StructuredObjectCreator is separated with a separator.

A StructuredObjectCreator delivers an ObjectEvent whenever the structure of the StructuredObjectCreator changes. Clients can add themselves as StructuredObjectCreatorListeners to receive notification when the structure of an individual key changes.

For information about using `StructuredObjectCreators`, see "Creating Groups of Buttons from a StructuredObjectCreator" on page 171.

G2 Palettes and G2 Object Creators

A `G2Palette` is a subclass of `Palette` for creating G2 items from a single G2 connection. To create a `G2Palette`, you:

- Provide a connection and a title string as arguments to the constructor.
- Add instances of a `G2ObjectCreator`, using versions of the `add` method, where each `G2ObjectCreator` creates a group, a hierarchy, or a dynamically updating set of `PaletteButtons` from a single G2 connection.

`G2ObjectCreator` implements the `StructuredObjectCreator` interface, which means it supports individual palette buttons or groups of palette buttons.

`G2ObjectCreator` provides methods for setting the following properties of the item representation on a `G2Palette`:

- Availability
- Structure

A `G2ObjectCreator` generates an `ObjectCreatorEvent` when:

- The G2 icon changes.
- The G2 class name changes.
- The G2 class is deleted.

When any of these events occur, the item representation on the `G2Palette` becomes unavailable, and the `G2ObjectCreator` notifies registered `ObjectCreatorListeners`.

For information about creating palettes of G2 objects, see "Creating G2 Palettes" on page 179.

GFR Palettes

Your G2 application might already contain G2 Foundation Resources (GFR) palettes. You can use the `GFRPalette` class in the `com.gensym.clscupgr.gfr` package to convert your GFR palettes directly into native palettes. To create a `GFRPalette`, you:

- Obtain the GFR palette KB workspace from a connection.
- Provide a title, a connection, and the GFR palette workspace as arguments to the constructor.

For information about creating palettes of G2 objects, see "Creating GFR Palettes" on page 181.

Comparing Palettes to Menus and Toolbars

A `Palette` is analogous to a `CMenu`, `CMenuBar`, `CPopupMenu`, or `ToolBar`, and an `ObjectCreator` is analogous to a `Command` as follows:

You add implementations of a(n)...	To this type of command-aware container...
ObjectCreator	Palette
Command	CMenu, CMenuBar, CPopupMenu, or Toolbar

Similarly, a `G2ObjectCreator` is analogous to an `AbstractStructuredCommand` as follows:

This class...	Provides a default implementation of this interface...	Which specifies these features...	And handles event notification of this listener...
G2ObjectCreator	Structured ObjectCreator	Key, description, icon, and availability of the item representation	Structured ObjectCreator Listener
Abstract Structured Command	Structured Command	Key, description, icon, availability, state, and immediacy	Structured Command Listener

The following table provides a summary of the classes you use to create palettes, and the classes you use to create menus and toolbars:

Palettes	Menus and Toolbars	Description
Palette G2Palette	CMenu CMenuBar CPopupMenu ToolBar	Containers
ObjectCreator StructuredObjectCreator	Command StructuredCommand	Interfaces that receive action events

Palettes	Menus and Toolbars	Description
`G2ObjectCreator`	`AbstractCommand` `AbstractStructuredCommand`	Default implementations of action event interfaces

`ObjectCreatorEvent`
`ObjectCreatorListener`
`StructuredObjectCreatorListener`
	`CommandEvent`
	`CommandListener`
	`StructuredCommandListener`
		Events and listeners for interfaces that receive action events

For background information on creating menus and toolbars from commands, see Chapter 5, "Creating Menus and Toolbars" on page 113.

Packages Covered

com.gensym.ntw.util

```
G2ObjectCreator
G2Palette
```

com.gensym.ui

Interfaces

```
ObjectCreator
ObjectCreator2
ObjectCreatorListener
ObjectFactory
RepresentationConstraints
StructuredObjectCreator
StructuredObjectCreatorListener
```

Classes

```
ObjectCreatorEvent
```

com.gensym.ui.palette

Interfaces

```
PaletteDropTarget
PaletteListener
```

Classes

```
Palette
PaletteButton
PaletteEvent
```

com.gensym.clscupgr.gfr

```
GFRPalette
```

Relevant Demos

This chapter shows the source code for the class located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\ palettedemo\PaletteDemo.java`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/ palettedemo/PaletteDemo.java`

Creating a Palette of Objects

To create a palette of objects where each object is represented as a button, you perform these tasks:

- Create an instance of a `Palette`.
- Create palette buttons from an `ObjectCreator`, `StructuredObjectCreator`, or `PaletteButton`.
- Add buttons to the palette by calling versions of the `add` method.
- Specify the palette behavior and layout.
- Handle `Palette` and `ObjectCreator` events.
- Get and set palette properties, as needed.

Creating the Palette

You can create a palette with "Palette" as the default title or with a title string that you supply.

To create a generic palette with a title:

➔ Create an instance of the `com.gensym.ui.palette.Palette` class, using this constructor:

```
Palette(String name)
```

Creating Palette Buttons

You can create palette buttons by using:

- `ObjectCreator`, which creates a `PaletteButton` for each key, by creating a `java.awt.Image`.
- `ObjectCreator2`, which creates a `PaletteButton` for each key, by creating a `javax.swing.Icon`.
- `StructuredObjectCreator`, which creates a tree-like structure where each leaf in the tree is a key and creates a `PaletteButton`.
- `PaletteButton`, which creates a palette button directly.

The instructions that follow use `ObjectCreator` to mean either an `ObjectCreator` or an `ObjectCreator2`.

Creating Buttons from an ObjectCreator

Each key in an `ObjectCreator` has:

- A short description, which the palette uses to represent the `PaletteButton` as text.
- A long description, which the palette uses as a tool tip when the cursor lingers over the button.
- A small icon and a large icon, which the palette uses to represent the `PaletteButton` as an icon.

`ObjectCreator` defines these final static variables to represent the textual description and the icon size:

```
LONG_DESCRIPTION
SHORT_DESCRIPTION
SMALL_ICON
LARGE_ICON
```

ObjectCreator extends ObjectFactory, which provides a method for creating the object associated with a particular key. The ObjectCreator creates the object when the user activates the PaletteButton by double-clicking.

To create one or more palette buttons:

➔ Create a class that implements this interface:

```
com.gensym.ui.ObjectCreator
```

The ObjectCreator class must implement these abstract methods:

Method	Description
getDescription(String key, int type)	Returns a textual description for the item specified by key, where type is LONG_DESCRIPTION or SHORT_DESCRIPTION.
getIcon(String key, int size)	Returns an image of the specified size for the item specified by key, where size is SMALL_ICON or LARGE_ICON.
getKeys()	Returns a String array that represents the keys.
createObject(String key)	Returns an object for the item specified by key.

Creating Groups of Buttons from a StructuredObjectCreator

A StructuredObjectCreator is an interface that defines a tree-like structure of PaletteButtons. The structure consists of an Object array, where each element in the array can be:

- A String[], where each array is a logical grouping of items. When added to a palette, the groups are visually separated by a space.
- An Object[], where each array defines a step in the hierarchy. When added to a palette, the array creates a subpalette.

Each String is a key that represents an item, where the leaves of the tree are always strings.

To create a tree-like structure of palette buttons:

➔ Create a class that implements this interface:

```
com.gensym.ui.StructuredObjectCreator
```

The `StructuredObjectCreator` class must implement this abstract method:

Method	Description
`getStructuredKeys()`	Returns an `Object[]` that defines the tree-like structure.

Creating Buttons Explicitly

A `PaletteButton` is a class that creates an individual item when the user double-clicks the button in a palette. You specify the object to create by using an `ObjectFactory`.

To create a palette button explicitly:

1. Create an instance of this class:

    `com.gensym.ui.palette.PaletteButton`

2. Create an instance of this class:

    `com.gensym.ui.ObjectFactory`

3. Call this method on the `ObjectFactory` to specify the object to create:

    `createObject(String key)`

    The `key` argument is the key associated with the `PaletteButton`.

4. Create the object by calling this method on the `PaletteButton`:

    `createObject()`

This method returns the object to create.

Adding Buttons to the Palette

You add buttons to a palette by calling different versions of the `add` method. You can add:

- Implementations of the `ObjectCreator` interface, which creates a `PaletteButton` for:
    - All command keys.
    - Individual command keys.
    - All command keys or a single command key with representation constraints.
- Any graphical element, such as a `PaletteButton`.

When you add an `ObjectCreator`, the icon of the `ObjectCreator` is used for the icon of the palette button. The size of the button is determined by the `iconSize` property of the `Palette`.

If your container includes several logical groupings of buttons, you can add a separator, as needed. Alternatively, you can add a `StructuredObjectCreator`, which creates its own separators.

For information on...	See...
`ObjectCreator`	"Creating Buttons from an ObjectCreator" on page 170.
`StructuredObjectCreator`	"Creating Groups of Buttons from a StructuredObjectCreator" on page 171.
`PaletteButton`	"Creating Buttons Explicitly" on page 172.

Adding All Keys of an ObjectCreator

The simplest way to add an `ObjectCreator` to a `Palette` is to create palette buttons for all of its keys.

To add an `ObjectCreator` with all of its keys to a palette:

➔ Call this version of the `add` method on a `Palette`:

```
add(ObjectCreator objectCreator)
```

The argument to the `add` method is an instance of an implementation of the `ObjectCreator` interface.

Adding Individual Keys of an ObjectCreator

You can create palette buttons for individual keys of an `ObjectCreator`.

To add an individual `ObjectCreator` key to a palette:

➔ Call this version of the `add` method on a `Palette`:

```
add(ObjectCreator objectCreator, String key)
```

The `objectCreator` argument is the same as described in "Adding All Keys of an ObjectCreator" on page 173.

The `key` argument is a string that represents the palette button to add.

Adding ObjectCreators with Representation Constraints

You can create palette buttons for all keys or individual keys of an `ObjectCreator`, using `RepresentationConstraints`, which lets you represent palette buttons as:

- Text only.
- Icon only.
- Text and icon.

If you choose to represent the button as both text and an icon, you can also specify the vertical and horizontal alignment, and the position of the text relative to the icon.

To add an `ObjectCreator` with representation constraints:

1. Create an instance of this class:

   ```
   com.gensym.ui.RepresentationConstraints
   ```

2. Specify the first argument to the constructor as one of the following final static variables:

   ```
   ICON_ONLY
   TEXT_ONLY
   TEXT_AND_ICON
   ```

3. If you choose to represent the object that the `ObjectCreator` creates as both text and icon, you can specify these additional arguments, in this order, in the `RepresentationConstraints` constructor:

   - `int horizontalAlignment` — Horizontal alignment of the text and icon relative to the palette.
   - `int verticalAlignment` — Vertical alignment of the text and icon relative to the palette.
   - `int horizontalTextPosition` — Horizontal position of the text relative to the icon.
   - `int verticalTextPosition` — Vertical position of the text relative to the icon.

   You specify these constraints by using the following final static variables, as needed:

   ```
   TOP
   BOTTOM
   RIGHT
   LEFT
   CENTER
   ```

4 Call either of these versions of the `add` method on a `Palette`, depending on whether you want to add all keys or a single key:

```
add(ObjectCreator objectCreator,
    RepresentationConstraints constraints)
```

```
add(ObjectCreator objectCreator,
    String key,
    RepresentationConstraints constraints)
```

The `objectCreator` and `key` arguments are the same as described in “Adding Individual Keys of an ObjectCreator” on page 173.

Adding Palette Buttons Directly

Rather than using an `ObjectCreator` to create palette buttons implicitly, you can add instances of a `PaletteButton` directly to a `Palette`.

To add an individual `PaletteButton` to a palette:

➔ Call this version of the `add` method on a `Palette`, passing a `PaletteButton` as the `component` argument:

```
add(Component component)
```

Adding Separators

If your palette includes groups of palette buttons, you can add a fixed space between those groups.

To add a separator between palette buttons:

➔ Call this version of the `add` method on a `Palette`:

```
addSeparator()
```

Specifying Palette Behavior and Layout

You can control the following features of a `Palette`’s behavior and layout:

- The default image that the palette uses for its buttons.
- The default icon size.
- The palette name.
- The orientation of the buttons in the palette.
- Whether the palette uses sticky mode when the user double-clicks a button.

Specifying the Default Image and Icon Size

A `Palette` uses the default image when the `getIcon` method of an `ObjectCreator` returns `null`.

To specify the default image:

➔ Call this method on a `Palette`, providing a `java.awt.Image` as argument:

```
setDefaultImage(Image image)
```

By default, a `Palette` uses large icons for its palette buttons. The `ObjectCreator` interface defines these two final static variables, which are integers, to define icon size:

```
SMALL_ICON
LARGE_ICON
```

To specify the default icon size:

➔ Call this method on a `Palette`:

```
setIconSize(int iconSize)
```

The `int` argument is one of the variables named above.

Specifying the Orientation of the Palette Buttons

By default, a `Palette` adds buttons horizontally to a palette. The `Palette` class defines these two final static variables, which are integers, to define the orientation:

```
HORIZONTAL
VERTICAL
```

To specify the orientation of the palette buttons:

➔ Call this method on a `Palette`:

```
setOrientation(int orientation)
```

The `int` argument is one of the variables named above.

Specifying the Behavior when the User Clicks a Palette Button

By default, when the user double-clicks a palette button, the button is released after the user clicks in a container to drop the item.

You can choose to have your palette use "sticky" mode, where the toggle button remains pressed until the user explicitly toggles it off.

To specify the behavior when the user clicks a palette button:

➔ Call this method on a `Palette`:

```
setStickyMode(boolean mode)
```

Getting Palette Properties

You can get the properties of a `Palette`. You can also get a list of all instances of the `Palette` class for a given application.

To get the properties of a palette:

➔ Call one of the following methods on a `Palette`:

```
getIconSize()
getName()
getOrientation()
isStickyMode()
```

To get a list of all palettes:

➔ Call this method on any `Palette`:

```
getPalettes()
```

Listening for Palette Events

Clients can implement the following listeners, located in the `com.gensym.ui.palette` package, to receive notification of `PaletteEvents`:

This listener...	Implements this method...	Which is called when...
`PaletteListener`	`paletteCreated(PaletteEvent)`	A `Palette` is created.
`PaletteDropTarget`	`paletteButtonStateChanged (PaletteEvent)`	A `PaletteButton` is toggled either on or off.

Notifying the Palette when the Drop is Complete or Cancelled

An implementation of the `PaletteDropTarget` needs to notify the `Palette` when the drop is completed or cancelled.

To notify a palette when the drop is complete:

➔ Call this method on a `Palette`:

```
dropOccurred()
```

This method resets the palette after a drop has occurred.

To notify a palette when the drop is cancelled:

➔ Call this method on a `Palette`:

```
dropCancelled()
```

This method cancels the pending drop.

Getting the Button that was Toggled

You might need to get the button or key associated with the `PaletteButton` that the user toggled.

To get the button that was toggled:

➔ Call this method on ac `PaletteEvent`:

```
getButton()
```

If the `PaletteEvent` is a result of toggling a `PaletteButton`, then the palette button that was toggled is returned; otherwise, `null` is returned.

To get the key associated with the button that was toggled:

➔ Call this method on the `PaletteEvent`:

```
getKey()
```

If the `PaletteEvent` is a result of toggling a `PaletteButton`, then the key associated with the palette button that was toggled is returned; otherwise, `null` is returned.

Listening for ObjectCreator Property Changes

Clients can implement the following listeners, located in the `com.gensym.ui` package, to receive notification of an `ObjectCreatorEvent`:

This listener...	Implements this method...	Which is called when this property changes...
`ObjectCreatorListener`	`availabilityChanged`	Availability
	`descriptionChanged`	Description
	`iconChanged`	Icon
`StructuredObjectCreatorListener`	`structuredChanged`	Structure

Testing for Availability

You can test when the item associated with a particular key of an `ObjectCreator` is available.

To test whether a palette button is available:

➔ Call this method on an `ObjectCreator`:

```
isAvailable(String key)
```

Getting the Key that Triggered the Event

You can get the key associated with the `ObjectCreator` that triggered an `ObjectEvent`.

To get the key that triggered the event:

➔ Call this method on an `ObjectCreatorEvent`:

```
getKey()
```

Creating G2 Palettes

You create a `G2Palette` the same way you create a `Palette` except that you also provide a connection.

You add palette buttons to the `G2Palette` the same way you add them to a `Palette` except that you must add a `G2ObjectCreator`.

Creating the Palette

You create a palette of G2 objects for a particular connection to G2, with or without a title.

To create a palette of G2 objects:

➔ Create an instance of this class:

```
com.gensym.ntw.util.G2Palette
```

For example, this code fragment creates a palette with a title:

```
private com.gensym.ntw.TwGateway connection;

palette = new G2Palette(connection, "Item Palette");
```

Adding Objects to the Palette

When you add an `ObjectCreator` to a `G2Palette`, the palette checks to ensure that:

- Only instances of a `G2ObjectCreator` are added.
- The `connection` argument to the `G2Palette` and the `G2ObjectCreator` are the same.

Otherwise, you add a `G2ObjectCreator` to a `G2Palette` in the same way that you add an `ObjectCreator` to a `Palette`.

For details, see "Adding Buttons to the Palette" on page 172.

Creating Palette Buttons from G2 Objects

You create palette buttons for a `G2Palette` by adding `G2ObjectCreator`s to the palette. You pass as the argument:

- A connection, which is an implementation of this interface:

  `com.gensym.ntw.TwAccess`

- An `Object` array that is either:
  - A `Symbol` array, where each symbol represents a G2 class name, which creates a set of palette buttons.
  - An array of `Symbol` arrays, which creates a group of palette buttons.

`G2ObjectCreator` uses the G2 class name as both the short and long description of the item representation. The key is the G2 class name as a string.

`G2ObjectCreator` obtains the icon for the item representation as follows:

For item representations whose classes are subclasses of...	**The icon is...**
The G2 entity class	The icon description of the G2 class.
The G2 text-box class	A standard text image that represents the text.

A `G2ObjectCreator` throws this exception, which is part of G2 JavaLink:

`com.gensym.jgi.G2AccessException`

To create palette buttons from G2 objects:

➔ Create an instance of this class:

`com.gensym.ntw.util.G2ObjectCreator`

For example, the following code fragment adds two G2 classes named pump and tank to a `G2Palette`:

```
private G2Palette palette;
private com.gensym.ntw.TwGateway connection;
private static final Symbol PUMP_ = Symbol.intern("PUMP");
private static final Symbol TANK_ = Symbol.intern("TANK");

palette.add(new G2ObjectCreator(connection,
                                new Symbol[]{PUMP_, TANK_}));
```

Creating GFR Palettes

To create a palette of G2 objects from a GFR palette, you provide:

- A title.
- A connection, which is an implementation of this interface:

  `com.gensym.ntw.TwAccess`

- The GFR palette, which is an instance of this class:

  `com.gensym.classes.KbWorkspace`

You can get the GFR palette workspace by calling `getUniqueNamedItem` on a `com.gensym.jgi.G2Gateway`.

To create a palette from a GFR palette:

➔ Create an instance of this class:

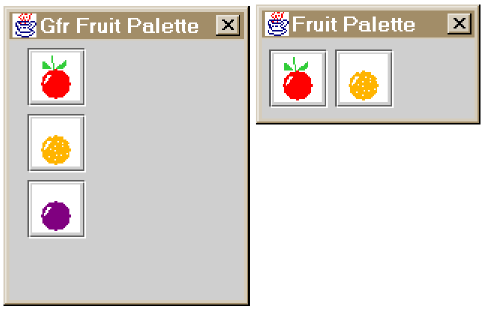
  `com.gensym.clscupgr.GFRPalette`

For example, this code fragment gets the GFR palette workspace and passes it in as an argument to the constructor, along with the connection:

```
private static final Symbol EXAMPLE_GFR_PALETTE_ =
    Symbol.intern("EXAMPLE-GFR-PALETTE");
private GFRPalette gfrPalette;
private com.gensym.ntw.TwGateway connection;

try{

    //Create GFRPalette
    KbWorkspace wksp = (KbWorkspace)connection.getUniqueNamedItem
        (KB_WORKSPACE_, EXAMPLE_GFR_PALETTE_);
    gfrPalette = new GFRPalette("GFR Fruit Palette", connection, wksp);
}
catch(G2AccessException e){
    e.printStackTrace();
}
```

Example

This example creates two Fruit palettes, which the application launches when you make a connection to G2. One is an instance of a `GFRPalette`, and the other is an instance of a `G2Palette`:




```
package com.gensym.demos.palettedemo;

import java.awt.*;
import java.awt.event.ActionEvent;
import com.gensym.shell.util.*;
import com.gensym.shell.commands.*;
import com.gensym.mdi.*;
import com.gensym.ui.*;
import com.gensym.ui.menu.*;
import com.gensym.ui.palette.*;
import com.gensym.ntw.TwAccess;
import com.gensym.ntw.TwConnectionListener;
import com.gensym.ntw.TwConnectionAdapter;
import com.gensym.ntw.TwConnectionEvent;
import com.gensym.jgi.G2AccessException;
import com.gensym.util.Symbol;
import com.gensym.ntw.util.G2Palette;
import com.gensym.ntw.util.G2ObjectCreator;
import com.gensym.message.Trace;
import com.gensym.clscupgr.gfr.*;
import com.gensym.classes.KbWorkspace;
import com.gensym.util.symbol.*;

public class PaletteDemo extends TW2Application
    implements G2ClassSymbols {

  private static final Symbol APPLE_ = Symbol.intern("APPLE");
  private static final Symbol ORANGE_ = Symbol.intern("ORANGE");
  private static final Symbol EXAMPLE_GFR_PALETTE_ =
    Symbol.intern("EXAMPLE-GFR-PALETTE");

  private MDIFrame frame;
  private TwAccess currentConnection;
  private TwConnectionListener loginListener;
```

```
  private WorkspaceCommands wkspHandler;
  private ExitCommands exitHandler;

  //G2Palette variables
   private G2Palette palette;
   private Dialog fruitPalette;

   //GFRPalette variables
   private GFRPalette gfrPalette;
   private Dialog gfrFruitPalette;

public PaletteDemo(String[] cmdLineArgs){
    super(cmdLineArgs);

    frame = new MDIFrame("Palette Demo");
    setCurrentFrame(frame);

    CMenuBar menubar = new CMenuBar();
    CMenu fileMenu = new CMenu("File");
    fileMenu.add(wkspHandler = new WorkspaceCommands(frame,
                                                  currentConnection));
    fileMenu.add(exitHandler = new ExitCommands(frame,
                                                currentConnection));
    menubar.add(fileMenu);
    CMenu g2Menu = new CMenu("G2");
    g2Menu.add(new ConnectionCommands(this));
    menubar.add(g2Menu);
    frame.setDefaultMenuBar(menubar);

    loginListener = new LoginAdapter();
    fruitPalette = new Dialog(frame, "Fruit Palette", false);
    gfrFruitPalette = new Dialog(frame, "Gfr Fruit Palette", false);

    frame.setSize(400, 300);
    frame.setVisible(true);
  }


  public ConnectionManager getConnectionManager(){
    return null;
  }

  public TwAccess getConnection(){
    return currentConnection;
  }
```

```
public void setConnection(TwAccess connection){
  if (connection == null)
    setConnection0(connection);
  else{
    if (connection.isLoggedIn())
  setConnection0(connection);
    else
  connection.addTwConnectionListener(loginListener);
  }
}

 private void setConnection0(TwAccess connection){
    if (currentConnection != null)
       currentConnection.removeTwConnectionListener(loginListener);
    currentConnection = connection;
    wkspHandler.setConnection(connection);
    exitHandler.setConnection(connection);
    if (connection != null){

       //Create G2Palette
       try{
          palette = new G2Palette(connection, "Fruit Palette");
          palette.add(new G2ObjectCreator
              (connection, new Symbol[]{APPLE_, ORANGE_}));
          fruitPalette.add(palette, BorderLayout.CENTER);
          fruitPalette.setVisible(true);
          fruitPalette.setSize(150, 80);
       }
       catch(G2AccessException e){
          e.printStackTrace();
       }
       try{

          //Create GFRPalette
          KbWorkspace wksp =
              (KbWorkspace)connection.getUniqueNamedItem
                  (KB_WORKSPACE_, EXAMPLE_GFR_PALETTE_);
          gfrPalette = new GFRPalette("GFR Fruit Palette",
                                      connection, wksp);
          gfrPalette.setOrientation(Palette.VERTICAL);
          gfrFruitPalette.add(gfrPalette, BorderLayout.CENTER);
          gfrFruitPalette.setVisible(true);
          gfrFruitPalette.setSize(60, 200);
       }
       catch(G2AccessException e){
          e.printStackTrace();
       }
    }
 }
```

```
class LoginAdapter extends TwConnectionAdapter{
    public void loggedIn(TwConnectionEvent event){
      TwAccess connection = (TwAccess)event.getSource();
      setConnection0(connection);
    }

    public void loggedOut(TwConnectionEvent event){
    }
  }

  public static void main(String[] args){
    PaletteDemo demo = new PaletteDemo(args);
  }
}
```

7

Creating Multiple Document Interface Containers

Describes how to create the various components of an MDI application, which include frames, child documents, and toolbar panels. Describes how to add documents to a frame, manage open documents, handle event notification, and create tiling commands for arranging documents in a frame.

Introduction

The `com.gensym.mdi` package provides the following containers, managers, events, and listeners, which you can use to create a multiple document interface (MDI) application:

- `MDIFrame` — A multiple document interface frame for displaying subclasses of `MDIDocument`.
- `MDIDocument` — A child frame of an `MDIFrame` in which you display views of the G2 server's data.
- `MDIManager` — A class that manages multiple documents in an `MDIFrame` and handles event notification.
- `MDIEvent` and `MDIListener` — An event that gets delivered when an `MDIManager` adds a document to an `MDIFrame`, and a listener for those events.
- `MDIToolBarPanel`— A container for displaying one or more toolbars in an `MDIFrame`.

For information on creating MDI applications that provide support for connecting to G2, see "TW2MDIApplication" on page 232.

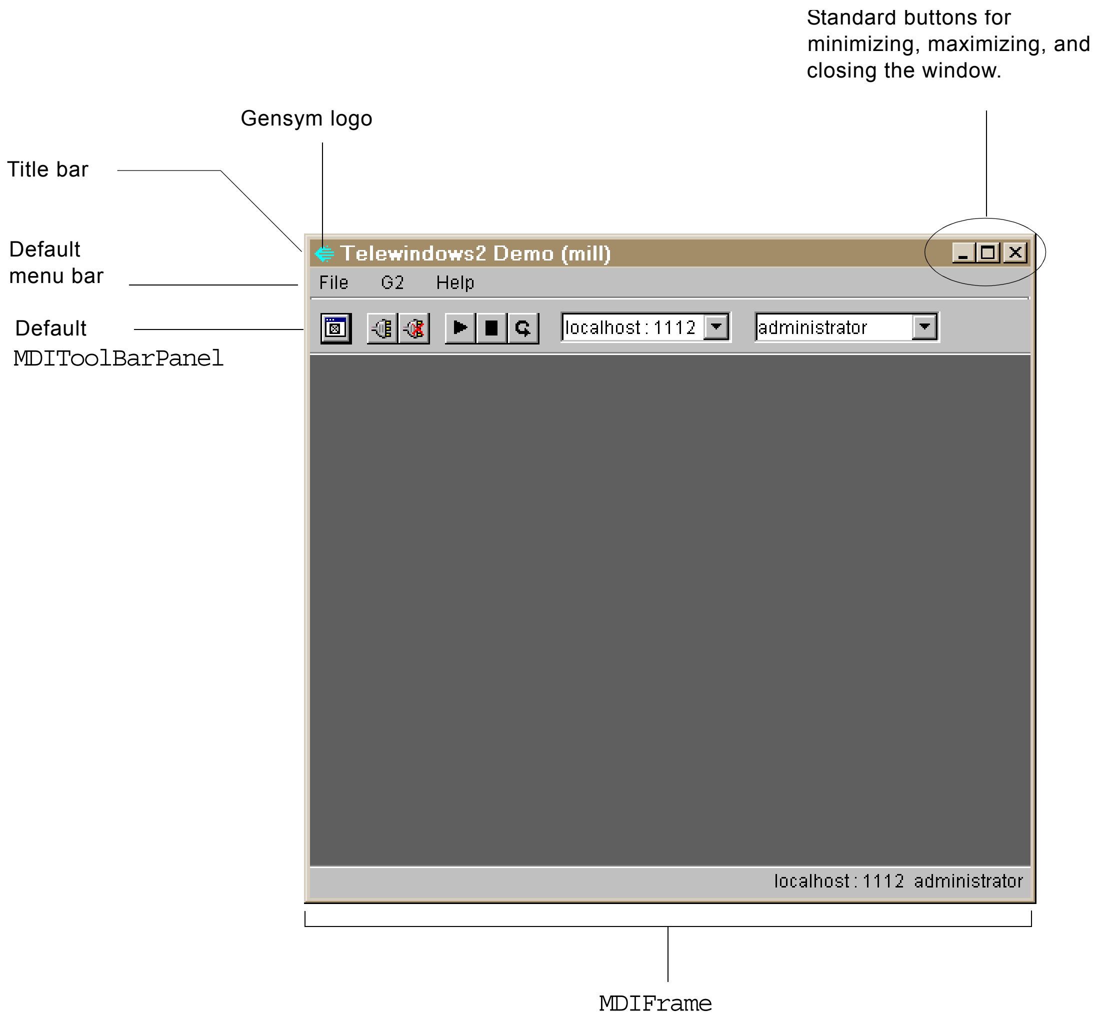
MDIFrame

An `MDIFrame` is a `javax.swing.JFrame` that provides methods for getting and setting the:

- Default menu bar.
- Default `MDIToolBarPanel`.
- Default Window menu.
- `MDIManager` for the frame.

You can create an MDIFrame with or without a title, default menu bar, default toolbar panel, and default Window menu.

Here is the default MDIFrame for the Telewindows2 (TW2) Toolkit default application shell before a connection has been made:



For details, see:

- "Creating the Frame" on page 193.
- Chapter 5, "Creating Menus and Toolbars" on page 113.
- "Creating an MDI Toolbar Panel" on page 197.
- javax.swing.JFrame

MDIDocument

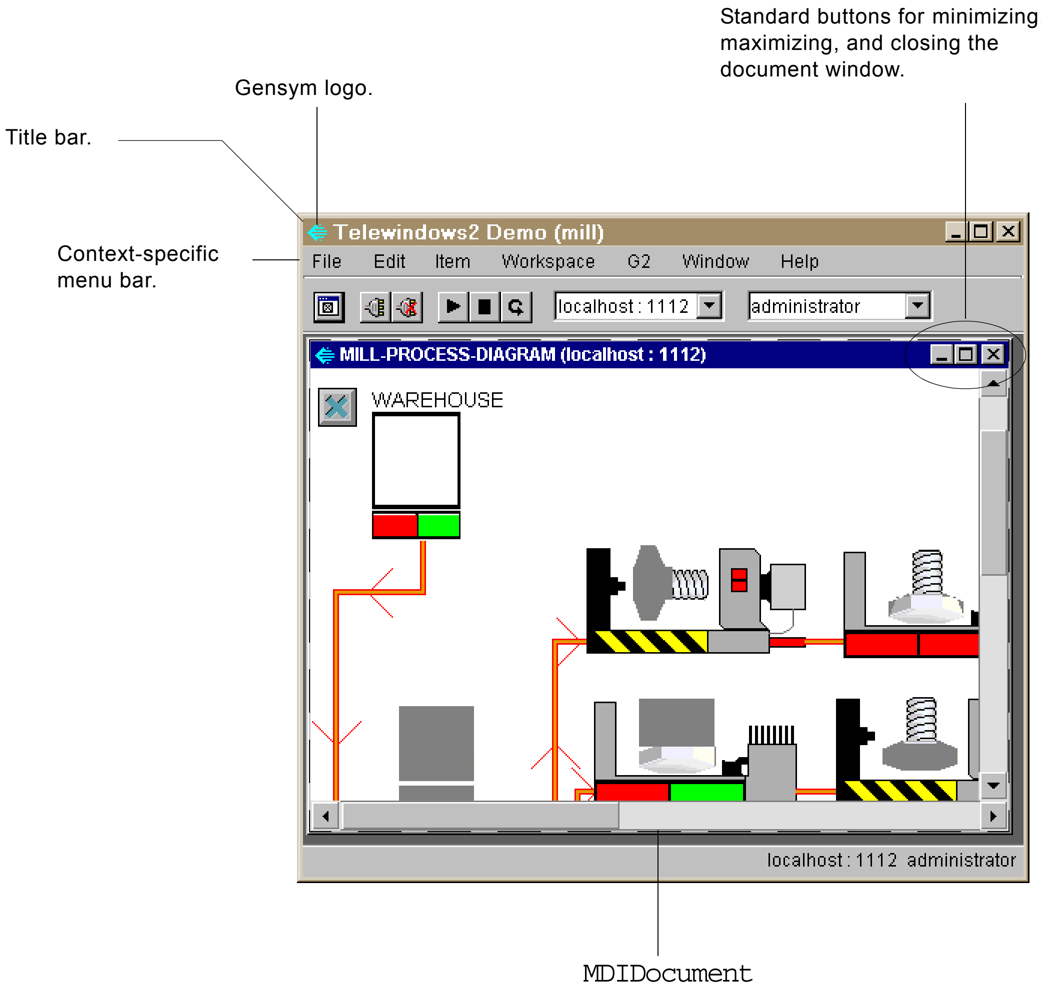
An `MDIDocument` is a child frame of an `MDIFrame` in which you display views into your G2 application's data. `MDIDocument` is an abstract class that you must extend to create your own `MDIDocument` type. Your MDI application can use one or more `MDIDocument` types to display different types of data.

An `MDIDocument` is a `javax.swing.JInternalFrame` that lets you create a child document with one or more of the following features:

- A title.
- A context-specific menu bar.
- A context-specific `MDIToolBarPanel`, which is located below the menu bar.
- A Window menu, which the `MDIManager` maintains.
- Standard buttons for minimizing, maximizing, resizing, and closing the document window.

The context-specific menu bar and toolbar panel get swapped in when the child document gains focus.

Here is the TW2 Toolkit shell when a child document has focus, where the context-specific toolbar is the same as for the default frame:



For details, see:

- "Adding Documents to the Frame" on page 199.
- "Creating MDI Document Types" on page 206.
- `javax.swing.JInternalFrame`.

MDIManager

An `MDIFrame` uses an `MDIManager` to add instances of `MDIDocuments` to the frame and to manage those documents. The `MDIManager` is responsible for:

- Maintaining a list of currently open documents.
- Maintaining the active document and the next document.

- Adding new documents to the frame, and determining the default size and location of those documents.
- Providing a built-in command for arranging documents vertically, horizontally, or in a cascade.
- Swapping context-specific menu bars for specific `MDIDocument` types.
- Handling event notification by generating an `MDIEvent` when an `MDIDocument` gets added to the frame.

The `MDIManager` implements the `MDITilingConstants` interface and returns commands for arranging documents in the frame.

For more information on...	See...
Getting the `MDIManager` and `MDIFrame`	• "Getting the Manager" on page 196. • "Getting the Frame" on page 196.
Managing documents	• "Getting Active and Open Documents" on page 200. • "Activating Documents" on page 202.
Event notification	"Listening for MDI Events" on page 204.
Using tiling commands	"Using Tiling Commands to Arrange Documents" on page 202.

Packages Covered

com.gensym.mdi

Interfaces

`MDIListener`
`MDITilingConstants`

Classes

`MDIEvent`
`MDIFrame`
`MDIManager`
`MDIToolBarPanel`

Relevant Demos

The following demos create and manipulate TW2 Toolkit MDI containers:

- `singlecxnmdiapp`
- `multiplecxnmdiapp`

The demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/`

Creating and Managing MDI Frames

You can create an MDI application by creating an instance of this class:

```
com.gensym.mdi.MDIFrame
```

Every `MDIFrame` has an associated manager, which is an instance of this class:

```
com.gensym.mdi.MDIManager
```

You get the `MDIManager` from the `MDIFrame`.

Creating the Frame

When you create an `MDIFrame`, you provide:

- A text string for the frame's title, which you can localize.
- A default menu bar.
- A Window menu for displaying all open documents.
- A default `MDIToolBarPanel`.

The following examples use resources to localize text. For more information on using resources, see Appendix A, "Localization."

Creating an MDIFrame with a Title

When you create an `MDIFrame` with just a title, you are responsible for setting the default menu bar and toolbar by calling methods on the `MDIFrame`.

For information on setting the default menu bar and toolbar panel, see "Setting the Default UI Controls of the Frame" on page 195.

To create an `MDIFrame` with a title:

➔ Call the `MDIFrame` constructor with a text string:

```
MDIFrame(String title)
```

For example, this code fragment creates an application frame named Workspace Browser:

```
MDIFrame appFrame = new MDIFrame("Workspace Browser")
```

Localizing the Title Bar Text of the MDIFrame

You can provide a localized text string as the title by creating a resource and providing a key.

To localize the title bar of an `MDIFrame`:

➔ Call the `MDIFrame` constructor with a localized text string as its argument.

To do this, you can call `getString` on a `Resource`, providing the key as its argument.

For example, this code fragment creates an application frame whose title is the localized text string associated with the `Title` key located in the `i18nUI` resource properties file:

```
private com.gensym.message.Resource i18nUI;

MDIFrame appFrame =
   new MDIFrame(i18nUI.getString("Title"));
```

Creating an MDIFrame with a Default Menu Bar and Toolbar Panel

You can provide a default menu bar and a default `MDIToolBarPanel` in the constructor for the `MDIFrame`. The `MDIFrame` displays the default menu bar and toolbar panel when no `MDIDocument` has focus.

If you provide a default menu bar and toolbar panel when you create the frame, you must also provide the menu in which the frame displays the list of currently open documents, which is typically the Window menu. If your application does not provide a Window menu, pass `null` for that argument.

When you create an `MDIFrame` by using this constructor, you must also set the default menu bar and tool bar panel, as described in “Setting the Default UI Controls of the Frame” on page 195.

For information on creating a menu bar, see “Creating Command-Aware Containers” on page 122.

For information on creating a toolbar panel, see “Creating an MDI Toolbar Panel” on page 197.

To create an `MDIFrame` with a default menu bar and toolbar panel:

➔ Call this `MDIFrame` constructor:

```
MDIFrame(String title,
        JMenuBar mb,
        JMenu windowMenu,
        MDIToolBarPanel tb)
```

For example, the following code fragment creates an `MDIFrame` with a localized title, a default menu bar, a Window menu, and a default toolbar panel:

```
private com.gensym.ui.menu.CMenuBar defaultMenuBar;
private com.gensym.ui.menu.CMenu windowMenu;
private com.gensym.ui.toolbar.ToolBar defaultToolBar;
private com.gensym.message.Resource i18nUI;

MDIFrame appFrame =
   new MDIFrame(i18nUI.getString("Title"),
               defaultMenuBar, windowMenu, defaultToolBar);
```

Setting the Default UI Controls of the Frame

If you create an `MDIFrame` by specifying a default menu bar, a default toolbar panel, and a default Window menu in the constructor, you must set these UI controls to be the default controls for the frame.

You typically set the default UI controls in the application's constructor.

To set the default menu bar:

➔ Call this method on an `MDIFrame`:

```
setDefaultMenuBar(JMenuBar defaultMenuBar)
```

To set the default menu bar and Window menu:

➔ Call this method on an `MDIFrame`:

```
setDefaultMenuBar(JMenuBar defaultMenuBar,
                  JMenu windowMenu)
```

To set the default toolbar panel:

➔ Call this method on an `MDIFrame`:

```
setDefaultToolBarPanel(MDIToolBarPanel defaultToolBarPanel)
```

Example

Setting the Default Menu Bar and Toolbar Panel

The following method might appear in the constructor of your application to set the default menu bar and toolbar panel for the `MDIFrame`.

Each set method takes as its argument an instance of the appropriate type of UI control. The set methods create these instances dynamically by calling a user-defined create method, which creates each control.

```
private com.gensym.mdi.MDIFrame frame;

private void createUiComponents() {
   JMenuBar menubar = createMenuBar();
   MDIToolBarPanel toolbarPanel = createToolbarPanel();
   frame.setDefaultMenuBar(menuBar);
   frame.setDefaultToolBarPanel(toolbarPanel);
}
```

Getting the Manager

You can call methods on the `MDIManager` to perform a number of tasks, including:

- Adding documents to the `MDIFrame`.
- Adding clients as an `MDIListener`.

To get the `MDIManager` from the `MDIFrame`:

➔ Call `getManager` on an `MDIFrame`.

For examples of calling methods on an `MDIManager`, see “Creating and Managing MDI Documents” on page 199.

Getting the Frame

You can get the `MDIFrame` from the `MDIManager`, although typically, you have access to the `MDIFrame` when you create it.

To get the `MDIFrame` from the `MDIManager`:

➔ Call `getFrame` on an `MDIManager`.

Creating an MDI Toolbar Panel

An `MDIToolBarPanel` can contain one or more toolbars, where each toolbar can have one of more toolbar buttons.

To create a toolbar panel in an `MDIFrame`:

1 Create an instance of this class to create the panel:

```
com.gensym.mdi.MDIToolBarPanel
```

2 Create one or more instances of this class to create individual toolbars:

```
com.gensym.ui.toolbar.ToolBar
```

3 Add each toolbar to the toolbar panel by calling the `add` method on the panel, providing a toolbar as its argument.

Example

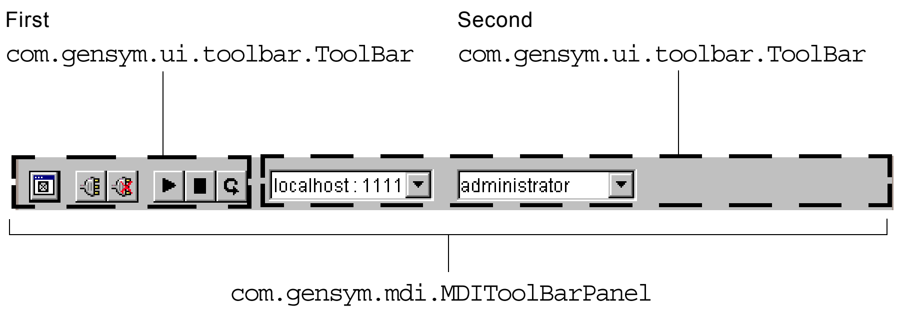
Creating an MDIToolbarPanel with Two Toolbars

The following example performs these tasks:

- Creates an instance of an `MDIToolBarPanel`.
- Creates an instance of a `ToolBar`.
- Adds toolbar buttons and separators to the toolbar.
- Creates another instance of a `ToolBar`.
- Adds instances of a `com.gensym.shell.util.HostPortPanel` and `com.gensym.shell.util.UserModelPanel` to the second toolbar.
- Adds the second toolbar to the toolbar panel.
- Returns the toolbar panel.

In the example, `G2AccessException` is in the `com.gensym.jgi` package, which is part of G2 JavaLink. See the API documentation for details.

This figure shows the toolbar panel and its toolbars:




```
private MDIToolBarPanel createToolBarPanel() {
   MDIToolBarPanel panel = new MDIToolBarPanel();
   ToolBar tb = new ToolBar();
   addWorkspaceCommandsToolBarButtons(tb);
   tb.addSeparator();
   addConnectionCommandsToolBarButtons(tb);
   tb.addSeparator();
   addG2StateCommandsToolBarButtons(tb);
   panel.add(tb);
   ToolBar tb2 = new ToolBar ();
   try {
      tb2.add (new HostPortPanel(connectionManager));
      tb2.add (javax.swing.Box.createGlue());
   } catch (G2AccessException e) {
      e.printStackTrace();
   }
   try {
      tb2.add (new UserModePanel(connectionManager, true));
   } catch (G2AccessException e) {
      e.printStackTrace();
   }
   panel.add(tb2);
   return panel;
}
```

For information about how to add toolbar buttons to toolbars, see “Creating Command-Aware Containers” on page 122.

Creating and Managing MDI Documents

Once you have created an `MDIFrame`, you call methods on its `MDIManager` to perform these tasks:

- Add `MDIDocuments` to the frame.
- Get the currently active document.
- Get a list of all open documents.
- Get a count of all open documents.
- Activate the next document in the array of open documents.

Adding Documents to the Frame

You add documents to the frame by creating an instance of a subclass of `MDIDocument` and calling a version of the `add` method on an `MDIManager`.

For information on getting the `MDIManager` from the `MDIFrame`, see “Getting the Manager” on page 196.

When you add a document to the frame, you can provide the dimensions and location of the new document, or you can use the default, which adds documents to the frame by arranging them in a cascade.

For information on overriding the default way in which the manager adds documents to the frame, see “Arranging New Documents” on page 203.

To add an `MDIDocument` to an `MDIFrame`:

1 Create an instance of a subclass of this class:

```
com.gensym.mdi.MDIDocument
```

2 Call `addDocument` on the `MDIManager`, providing an `MDIDocument` subclass as its first argument and, optionally, the following arguments or set of arguments:

   - An instance of a `java.awt.Dimension` object, which specifies the dimensions of the document.
   - The length and width of the document, and the x-offset and y-offset of the top-left corner of the document from the top-left corner of the frame, all expressed as integers.

The document gets added to the `MDIFrame` according to the dimension or offsets you specify.

Examples

The following examples gets the current frame from the application by calling a method on `com.gensym.core.UiApplication`, which is part of G2 JavaLink.

The examples use these variables:

```
private com.gensym.ntw.TwGateway twConnection;
private com.gensym.classes.KbWorkspace kbWorkspace;
```

Adding an MDIDocument to an MDIFrame

This code fragment adds a `SingleCxnMDIWorkspaceDocument`, which is a subclass of `MDIDocument`, to an `MDIFrame`:

```
SingleCxnMDIWorkspaceDocument wkspDoc =
   new SingleCxnMDIWorkspaceDocument(twConnection, kbWorkspace);
MDIFrame frame = (MDIFrame)UiApplication.getCurrentFrame();
MDIManager manager = frame.getManager();
manager.addDocument(wkspDoc);
```

Adding an MDIDocument of a Given Dimension to an MDIFrame

The following code fragment adds a `SingleCxnMDIWorkspaceDocument` of a given dimension to the `MDIFrame` associated with a `com.gensym.core.UiApplication`.

The example gets the `java.awt.Dimension` object by calling `getPreferredSize` on `MDIDocument`, which is a `javax.swing.JComponent`.

```
SingleCxnMDIWorkspaceDocument wkspDoc =
   new SingleCxnMDIWorkspaceDocument(twConnection, kbWorkspace);
MDIFrame frame = (MDIFrame)UiApplication.getCurrentFrame();
MDIManager manager = frame.getManager();
manager.addDocument(wkspDoc, wkspDoc.getPreferredSize());
```

Getting Active and Open Documents

The `MDIManager` provides methods for getting:

- The document that currently has focus, which is called the active document.
- The list of all open documents.
- The count of all open documents.

To get the currently active document:

➔ Call this method on an `MDIManager`:

```
getActiveDocument()
```

To get a list of open documents:

➔ Call this method on an `MDIManager`:

```
getDocuments()
```

To get the count of all open documents:

➔ Call this method on an `MDIManager`:

```
getDocumentCount()
```

Example

Getting All Open Documents and Getting the Active Document

The following example shows a constructor for a command that prints the current workspace. To do this, the command calls these methods on an `MDIManager`:

- `getDocuments`
- `getActiveDocument`

The constructor provides a single command key for printing the workspace view associated with the currently active document. The constructor is responsible for making the command key available based on whether a workspace document is in focus. It does this by adding the command as a `java.beans.PropertyChangedListener` so it receives notification when the currently active workspace document gains or looses focus.

To get the open documents and the currently active document, the method performs these tasks:

- Calls `getManager` on an `MDIFrame` to get the `MDIManager`.
- Calls `getDocuments` on the `MDIManager` to get an array of all instances of `MDIDocument` in the frame.
- Calls `getActiveDocument` on the `MDIManager` to get the currently active document.

For information on the arguments to `CommandInformation` and general information on creating commands, see “Creating Commands” on page 131.

Here is the constructor for a command that prints the current document:

```
private MDIFrame frame;

public PrintWorkspaceCommand(MDIFrame parentFrame){
    super(new CommandInformation[]{
        new CommandInformation(PRINT_WORKSPACE, true,
                               shortResource, longResource,
                               null, null, null, false)});
    if (parentFrame != null){
        frame = parentFrame;
        frame.getManager().addMDIListener(this);
        MDIDocument[] docs = frame.getManager().getDocuments();
        for (int i=0; i<docs.length; i++){
            if (docs[i] instanceof WorkspaceDocument)
                docs[i].addPropertyChangeListener(this);
        }
        MDIDocument activeDoc = frame.getManager().
getActiveDocument();
        setAvailable(PRINT_WORKSPACE,
            (activeDoc instanceof WorkspaceDocument));
    }
}
```

Activating Documents

You can activate the next document in the array of currently open documents to cycle through the available documents, making each successive document gain focus.

To make the next document become the active document:

➔ Call this method on an MDIManager:

```
activateNextDocument()
```

Using Tiling Commands to Arrange Documents

The MDIManager provides a standard tiling command for arranging MDIDocuments, which consists of these three actions:

- Cascade
- Tile Horizontally
- Tile Vertically

Getting the Default Tiling Commands

To get the default tiling commands, call a method on the MDIManager, then add the return value of the method to a command-aware container, such as a menu or toolbar.

For information on adding commands to command-aware containers, see "Creating Command-Aware Containers" on page 122.

To use the default tiling commands:

➔ Call this method on an MDIManager:

```
getTilingCommand()
```

For example, the following method creates a Window menu by adding tiling commands to a com.gensym.ui.menu.CMenu:

```
private MDIFrame frame;

private static CMenu createWindowMenu(){
    windowMenu = new CMenu("Window");
    windowMenu.add(frame.getManager().getTilingCommand());
    return windowMenu;
}
```

Arranging New Documents

By default, the MDIManager adds new documents to an MDIFrame in a cascade.

You can choose to arrange new MDIDocuments vertically or horizontally by calling a method on the MDIManager. You pass as the argument one of the static final variables that this interface provides, which MDIManager implements:

```
com.gensym.mdi.MDITilingConstants
```

The interface provides the following three static final variables, which are integers:

```
TILE_CASCADE
TILE_HORIZONTALLY
TILE_VERTICALLY
```

To customize the default arrangement when adding new documents to a frame:

➔ Call this method on an MDIManager and provide one of the static final variables that the MDITilingConstants interface defines:

```
arrange(int arrangementCode)
```

For example, this method arranges new MDIDocuments vertically:

```
frame.getManager().arrange(TILE_VERTICALLY);
```

Listening for MDI Events

You can add and remove clients as MDIListeners to receive notification when an MDIDocument gets added to an MDIFrame. The MDIListener interface defines the documentAdded method to determine the behavior of the client when a document gets added.

The MDIManager delivers an MDIEvent to registered listeners whenever it adds an MDIDocument to an MDIFrame.

You call getDocument on the MDIEvent to get the document that was added.

To listen for MDIEvents:

1 Create a class that implements this interface:

```
com.gensym.mdi.MDIListener
```

2 Add and remove clients as listeners by calling the appropriate methods on:

```
com.gensym.mdi.MDIManager
```

Example

Suppose you wanted to define a command that prints a workspace. The command would implement the MDIListener interface so it receives notification when a document gets added to the frame. The command would then set a com.gensym.wksp.ScalableWorkspaceView component into the document when it gets added.

The command would also implement the java.beans.PropertyChangedListener so it can make the command unavailable when the workspace document loses focus.

For information on the arguments to CommandInformation and general information on creating commands, see “Creating Commands” on page 131.

Implementing the MDIListener

Here is a class definition for PrintWorkspaceCommand, which listens for MDIEvents and PropertyChangedEvents:

```
public final class PrintWorkspaceCommand
   extends AbstractCommand
   implements MDIListener, PropertyChangeListener {
   //Additional code
}
```

Adding a Client as an MDIListener

Here is the constructor for the command, which adds itself as a listener for `MDIEvents`:

```
private MDIFrame frame;

public PrintWorkspaceCommand(MDIFrame parentFrame){
   super(new CommandInformation[]{
      new CommandInformation(PRINT_WORKSPACE, true,
                             shortResource, longResource,
                             null, null, null, false)});
   if (parentFrame != null){
      frame = parentFrame;
      frame.getManager().addMDIListener(this);
      //Additional code
   }
   //Additional code
}
```

Implementing the Behavior of the MDIListener

The following method provides the implementation of the `documentAdded` listener method for `MDIListener`, which performs these tasks:

- Gets the `MDIDocument` from the `MDIEvent` by calling `getDocument`.
- Tests to determine the type of `MDIDocument` that was added.
- Casts the type of the document that gets added to be a `com.gensym.shell.util.WorkspaceDocument`, which is a subclass of `MDIDocument`.
- Adds itself as a `PropertyChangedListener`.
- Sets the workspace view of the workspace document by calling a private method called `setWorkspaceView`, which takes an instance of a `com.gensym.wksp.WorkspaceView`.

```
public void documentAdded(MDIEvent event) {
   MDIDocument document = (MDIDocument)event.getDocument();
   if (document instanceof SingleCxnMDIWorkspaceDocument) {
      SingleCxnMDIWorkspaceDocument wkspDoc =
         (SingleCxnMDIWorkspaceDocument)document;
      wkspDoc.addPropertyChangeListener(this);
      setWorkspaceView(wkspDoc.getWorkspaceView());
   }
}
```

Creating MDI Document Types

The `com.gensym.shell.util` package provides two `MDIDocument` types that you can use in your application, depending on your needs:

- `TW2Document` — A generic `MDIDocument` associated with a connection to G2 to which you can add any view into the G2 server's data.
- `WorkspaceDocument` — A `TW2Document` that displays a `com.gensym.wksp.ScalableWorkspaceView` component to which you can add a context-specific menu bar.

For information on these document types, see Chapter 8, "Using Telewindows2 Toolkit MDI Documents" on page 207.

If neither of these `MDIDocument` types meets your needs, you can create a custom `MDIDocument` type. `MDIDocument` types can contain any view into your G2 server's data.

To create an `MDIDocument` type:

➔ Subclass this abstract class:

```
com.gensym.mdi.MDIDocument
```

8

Using Telewindows2 Toolkit MDI Documents

Describes the various MDI document types that you can use and extend to create documents that display workspace views and other views into your G2 server's data. Describes the associated factories that you can use and extend to generate different types of workspace documents.

Introduction

The `com.gensym.shell.util` package provides two `MDIDocument` types, which you can use to display views into the G2 server's data:

- `TW2Document` — A subclass of `MDIDocument` that displays a view into the G2 server's data, for example, an object manager or a module editor.
- `WorkspaceDocument`— A subclass of `TW2Document` that displays a `com.gensym.wksp.ScalableWorkspaceView` component and provides its own context-specific menu bar and `MDIToolBarPanel`.

An `MDIDocument` that displays a workspace view is called a **workspace document**.

The `com.gensym.shell.util` package also provides a factory for generating your own types of workspace documents, and a default implementation of that factory:

- `WorkspaceDocumentFactory` — An interface that you implement to generate your own type of workspace document. You create your own type of workspace document to provide a context-specific menu bar and toolbar panel.
- `DefaultWorkspaceDocumentFactoryImpl` — A default implementation of the `WorkspaceDocumentFactory` interface that generates an instance of a `WorkspaceDocument`.

A factory that generates any type of `WorkspaceDocument` is called a **workspace document factory**.

Packages Covered

com.gensym.shell.util

Interfaces

`WorkspaceDocumentFactory`

Classes

`DefaultWorkspaceDocumentFactoryImpl`
`TW2Document`
`WorkspaceDocument`

Relevant Demos

The demo in the following directory, depending on your platform, creates an MDI document type and factory:

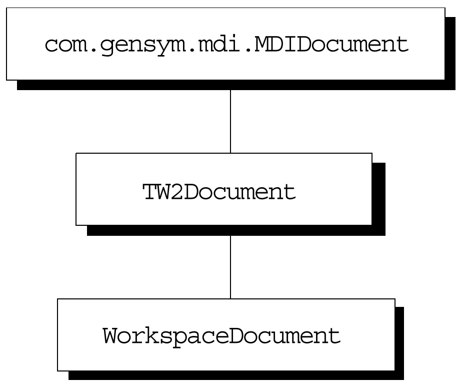
NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\singlecxnmdiapp`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/singlecxnmdiapp`

Using MDI Document Types

This section shows the inheritance structure for the `MDIDocument` types and describes the features, behaviors, and methods of each document type. It then describes how to create `MDIDocument` types that display views of G2 server data.

Class Hierarchy of MDIDocument Types

This figure shows the class hierarchy of the `MDIDocument` types in the `com.gensym.shell.util` package:



For information on `MDIDocument`, see “Creating and Managing MDI Documents” on page 199.

TW2Document

Features

`TW2Document` provides these features:

- A constructor that takes a connection as its argument.
- A constructor that takes a connection, a menu bar, a Window menu, and a `MDIToolBarPanel` as its arguments.

Behavior

`TW2Document` has this behavior:

- Uses the default menu bar and toolbar panel from the `MDIFrame`, if not provided in the constructor.
- Closes the document when the connection closes.
- If the document has an associated `com.gensym.shell.util.ConnectionManager`, makes the document inactive when the connection switches.

Methods

TW2Document supports these methods:

- getConnection — Gets the current connection in an application that supports single connections to G2.
- getConnectionManager — Gets the com.gensym.shell.util.ConnectionManager in an application that supports multiple connections to G2.

WorkspaceDocument

Features

WorkspaceDocument provides these features:

- A constructor that creates a workspace document with scroll bars, given a connection and a KB workspace.
- A constructor that creates a workspace document with scroll bars, given a connection, a KB workspace, a menu bar, a Window menu, and an MDIToolBarPanel as its arguments.

Behavior

WorkspaceDocument closes the document when the KB workspace is deleted in G2.

Creating MDI Documents that Display Views into the G2 Server's Data

You can create different MDIDocument types to display:

- Any view into the G2 server's data.
- A workspace view with a context-specific menu bar and MDIToolBarPanel.
- A workspace view that uses the default menu bar and MDIToolBarPanel of the MDIFrame.

To create an `MDIDocument` that displays a view into the G2 server's data:

1. Extend one of these classes, depending on the view you want the document to display:

To view...	Extend...
Any type of G2 server data	`com.gensym.shell.util.TW2Document`
KB workspaces	`com.gensym.shell.util.WorkspaceDocument`

2. Call one of the various constructors for the superior class to create an `MDIDocument` type, with one or more of the following features:
   - A context-specific menu bar.
   - A context-specific `MDIToolBarPanel`.
   - A Window menu.
3. Build the context-sensitive menu bar, as needed.

   For details, see "Creating Command-Aware Containers" on page 122.
4. Build the context-specific `MDIToolBarPanel`, as needed.

   For details, see "Creating an MDI Toolbar Panel" on page 197.

To create an `MDIDocument` that uses the default menu bar and toolbar panel:

➔ Create an instance of this class:

```
com.gensym.shell.util.WorkspaceDocument
```

Using Workspace Document Factories

Typically, your application needs to create its own type of `WorkspaceDocument` to provide a context-specific menu bar and toolbar panel that are applicable to your application. You use a factory to generate the desired type of workspace document. Each class that creates an instance of any type of `WorkspaceDocument` is responsible for calling a method that sets the workspace document factory. You call this method once for each class that creates a workspace document.

This table determines when you need to create a workspace document factory:

To generate...	You...
A `WorkspaceDocument` that uses the default menu bar and `MDIToolBarPanel` of the `MDIFrame`	Do not need to create a workspace document factory; the application uses a `DefaultWorkspaceDocumentFactoryImpl`.
A `WorkspaceDocument` subclass that defines a context-specific menu bar and `MDIToolBarPanel`	Must implement the `WorkspaceDocumentFactory` interface.

To generate workspace documents, using a factory:

1. Create a class that implements this interface:

   `com.gensym.shell.util.WorkspaceDocumentFactory`

2. Define a method on this class that returns an instance of a subclass of `WorkspaceDocument`.

   Typically, this method is called `createWorkspaceDocument`.

3. For each class that generates a type of `WorkspaceDocument`, create a method that takes as its argument an instance of your implementation of `WorkspaceDocumentFactory` and sets the current factory to this argument.

   Typically, this method is called `setWorkspaceDocumentFactory`.

4. In the constructor for the application:

   a. Create an instance of your implementation of `WorkspaceDocumentFactory`.

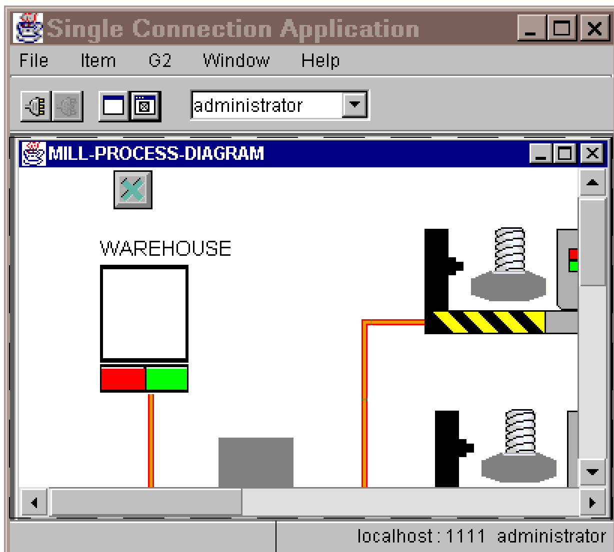
   b. For each class that generates a workspace document, call the method that sets the current factory, passing your implementation of `WorkspaceDocumentFactory` as the argument.

      You should only call this method once for each class that generates a workspace document.

Example

This example creates a subclass of `WorkspaceDocument` called `SingleCxnMdiWorkspaceDocument`, which provides a context-specific menu bar for a single connection MDI application.

The workspace document looks like this when displayed in the `MDIFrame`:



Context-specific menu bar

`SingleCxnMdiWorkspaceDocument`

The workspace document:

- Extends `WorkspaceDocument`.
- Initializes the `WorkspaceDocumentFactory`.
- Calls the constructor for the superior class, which:
  - Creates a context-specific menu bar.
  - Uses the default `MDIToolBarPanel` of the `MDIFrame`, which it gets from the `com.gensym.core.UiApplication`.
- Creates a context-specific menu bar and associated menus.

Creating a Custom Workspace Document

Here is the custom workspace document class:

```
public class SingleCxnMdiWorkspaceDocument extends WorkspaceDocument {

    //Private variables
    private static Resource i18nUI =
        Resource.getBundle("com.gensym.demos.singlecxnmdiapp.Messages");
    private static MDIFrame frame =
        (MDIFrame)MDIApplication.getCurrentFrame();
    private static CMenuBar menuBar = createMenuBar();
    private static CMenu windowMenu;
    private static TwAccess currentConnection;
    private static boolean alreadySetupConnectionCmds = false;

    //Create a SingleCxnMdiWorkspaceDocument for the specified
    //connection and workspace
    public SingleCxnMdiWorkspaceDocument(TwAccess connection,
                                         KbWorkspace wksp)
        throws G2AccessException{

            //Call constructor for superior class
            super(connection, wksp, menuBar, windowMenu,
                frame.getDefaultToolBarPanel());

            //Initialize current connection
            currentConnection = connection;
        }

    //Create context-specific menu bar
    private static CMenuBar createMenuBar(){
        menuBar = new CMenuBar();
        menuBar.add(createFileMenu());
        menuBar.add(createItemMenu());
        menuBar.add(createViewMenu());
        menuBar.add(createG2Menu());
        menuBar.add(createWindowMenu());
        menuBar.add(createHelpMenu());
        return menuBar;
    }

    //Create File menu
    private static CMenu createFileMenu() {
        //File menu
    }

    //Create Item menu
    private static CMenu createItemMenu(){
        //Item menu
    }
```

```
    //Create View menu
    private static CMenu createViewMenu(){
        //View menu
    }
    //Create G2 menu
    private static CMenu createG2Menu() {
        //G2 menu
    }
    //Create Window menu
    private static CMenu createWindowMenu(){
        //Window menu
    }
    //Create Help menu
    private static CMenu createHelpMenu() {
        //Help menu
    }
}
```

Implementing a Workspace Document Factory

This class implements a `WorkspaceDocumentFactory` by implementing a `createWorkspaceDocument` method, which returns an instance of a `SingleCxnMdiWorkspaceDocument`, a subclass of `WorkspaceDocument`:

```
public class SingleCxnMdiWorkspaceDocumentFactoryImpl
    implements WorkspaceDocumentFactory {
    public WorkspaceDocument createWorkspaceDocument
        (TwAccess connection, KbWorkspace workspace) {
        return new SingleCxnMdiWorkspaceDocument(connection,
                                                 workspace);
    }
}
```

Setting the Workspace Document Factory

In this example, the `WorkspaceCommandsImpl` class generates a workspace document when the user gets a KB workspace. Thus, it must:

- Define a method that sets the current workspace document factory.
- Set the workspace document factory in the application's constructor.

Here is the definition of the `setWorkspaceDocumentFactory` method on the `WorkspaceCommandImpl` class:

```
private WorkspaceDocumentFactory factory =
    new DefaultWorkspaceDocumentFactoryImpl();
private boolean wkspDocFactorySet = false;
private com.gensym.message.Resource i18n =
    Resource.getBundle("Errors");

public void setWorkspaceDocumentFactory (WorkspaceDocumentFactory
                                         factory){
    if (wkspDocFactorySet)
        throw new Error (i18n.getString
            ("WorkspaceDocumentFactoryAlreadyDefined"));
    else{
        this.factory = factory;
    wkspDocFactorySet = true;
    }
}
```

Here is the method that the application's constructor calls to set the workspace document factory for an instance of `WorkspaceCommandsImpl`:

```
private void registerWorkspaceDocumentFactory() {
    singleCxnMdiWkspDocFactory =
        new SingleCxnMdiWorkspaceDocumentFactoryImpl();
    if (wkspHandler != null)
        ((WorkspaceCommandsImpl)wkspHandler).
            setWorkspaceDocumentFactory(singleCxnMdiWkspDocFactory);
}
```

Application Classes

Creating Telewindows2 Toolkit Applications

Describes the application classes you can extend to create generic UI applications, SDI applications, and MDI applications, and describes the required and optional features of each. Describes how to create single and multiple connection applications, and how to implement the abstract methods that manage connections. Describes how to implement the specific features of SDI and MDI applications.

Introduction

Telewindows2 (TW2) Toolkit provides a number of classes that you can extend to help you build G2 client applications. These classes manage application frames and connections through their API, and allow users to view and manipulate G2 data.

Before you begin developing your TW2 Toolkit application, answer the following questions to determine the type of application you should create:

- Will the end user run the application through a user interface?
- Will the user interface support a way of making a connection to G2?
- Will the application provide a single document window or multiple document windows?
- Will the application support single or multiple connections to G2?

By answering these basic questions about your application, you can determine which class you should extend to create your application, and which class you should use to create and manage connections.

UI Applications

Most G2 client applications provide a user interface to support interacting with G2 items on workspaces. In its simplest form, a G2 client application provides an application frame with a visual representation of G2 data, typically a workspace view. However, the application frame can provide other views into the G2 server's data, as well.

G2 JavaLink supports these two classes for creating TW2 Toolkit applications:

- `GensymApplication` — Provides support for creating a generic G2 client application.
- `UiApplication` — Provides support for creating a generic G2 client application with a user interface, which manages the application frame through its API.

Because the TW2 Toolkit classes for creating applications inherit from these classes, this chapter explains both of these classes.

SDI and MDI Applications

Depending on your application, you might need to support multiple documents within the application window, or you might need to support only a single document. For example:

- Most modern word processors and spreadsheets support multiple document windows, which contain text documents or spreadsheets, respectively.
- Most Web browsers and some paint programs support a single document window, which contains a single Web page or graphic, respectively.

The com.gensym.shell.util package provides two classes that you can extend to create each of these types of applications:

- TW2Application — Provides support for creating a single document interface (SDI) application, which manages single and multiple connections to G2.
- TW2MDIApplication — Provides support for creating a multiple document interface (MDI) application, which manages single and multiple connections to G2.

Because both of these classes inherit from UiApplication, they also provide support for managing the application frame through their API.

The com.gensym.shell.util package also provides the following classes for creating and managing multiple connections to G2:

- ConnectionManager — Creates and manages multiple connections to the G2 server through the client.
- ContextChangeListener — Handles the events associated with multiple connections to G2.

Organization of this Chapter

The following table describes where to go in this chapter for information on creating Telewindows2 (TW2) Toolkit applications:

For information on...	See...
Answering the questions that help you determine the type of application you will create	“Determining Which Application Foundation Class to Extend” on page 223.
The features, behavior, and key methods of the classes you use for creating applications	“Application Foundation Classes” on page 227.
The required and optional features of TW2 Toolkit SDI or MDI applications	“Creating Telewindows2 Toolkit Applications” on page 233.
Creating and managing multiple connections to G2	“Creating and Managing Connections to G2” on page 236.

For information on...	See...
Specific features of TW2 Toolkit SDI applications	"Creating Single Document Interface Applications" on page 247.
Specific features of TW2 Toolkit MDI applications	"Creating Multiple Document Interface Applications" on page 251.

Packages Covered

com.gensym.shell.util

Interfaces

`ContextChangedListener`

Classes

```
ConnectionManager
ContextChangedEvent
TW2Application
TW2MDIApplication
TW2MDIWorkspaceShowingAdapter
TW2WorkspaceShowingAdapter
```

com.gensym.mdi

`MDIApplication`

com.gensym.core

```
GensymApplication
UiApplication
```

Relevant Demos

The following demos show examples of creating TW2 Toolkit applications:

- `wksppanel`
- `singlecxnsdiapp`
- `singlecxnmdiapp`
- `multiplecxnsdiapp`
- `multiplecxnmdiapp`

The demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/`

Determining Which Application Foundation Class to Extend

Telewindows2 Toolkit provides a number of **application foundation classes**, which are classes upon which you can build G2 client applications. To build an application, you extend the class that provides the features you need and implement its abstract methods.

To determine which application foundation class to extend, answer the questions in the following headings.

Will the Application Have a User Interface?

You can create a G2 client application that interacts with the server through its data or through a user interface. To create a G2 client application, extend one of the following application foundation classes:

To create an application that...	Extend this class...
Interacts with the G2 server through its data	`com.gensym.core.GensymApplication`
Interacts with the G2 server through a user interface	`com.gensym.ntw.util.UiApplication`

A G2 client application that interacts with the server through a user interface is called a **UI application**. The UI application is responsible for managing the application frame and its connections to G2.

Will the Application Support Connecting to G2 Through the UI?

If you are creating a UI application, you need to determine whether *you* want to manage connections to G2 as part of the application, or whether you want *the application* to handle those connections for you.

The `com.gensym.shell.util` package provides two application foundation classes that you can extend to provide built-in support for handling connections to G2:

To create an application that...	Extend this class...
Handles its own connections to G2	`com.gensym.core.UiApplication`
Provides built-in support for managing connections to G2	`com.gensym.shell.util.TW2Application` or `com.gensym.shell.util.TW2MDIApplication`

For general information on creating applications that manage connections to G2, see "Creating Telewindows2 Toolkit Applications" on page 233.

Will the Application Provide a Single or Multiple Document Frame?

You can create one of these two types of applications, both of which manage connections to G2 through their API:

- **Single document interface (SDI) application**, which contains a single frame in which to display and manipulate G2 data.
- **Multiple document interface (MDI) application**, which contains multiple child frames, or documents, for displaying and manipulating G2 data.

To create these types of applications, extend one of the following application foundation classes:

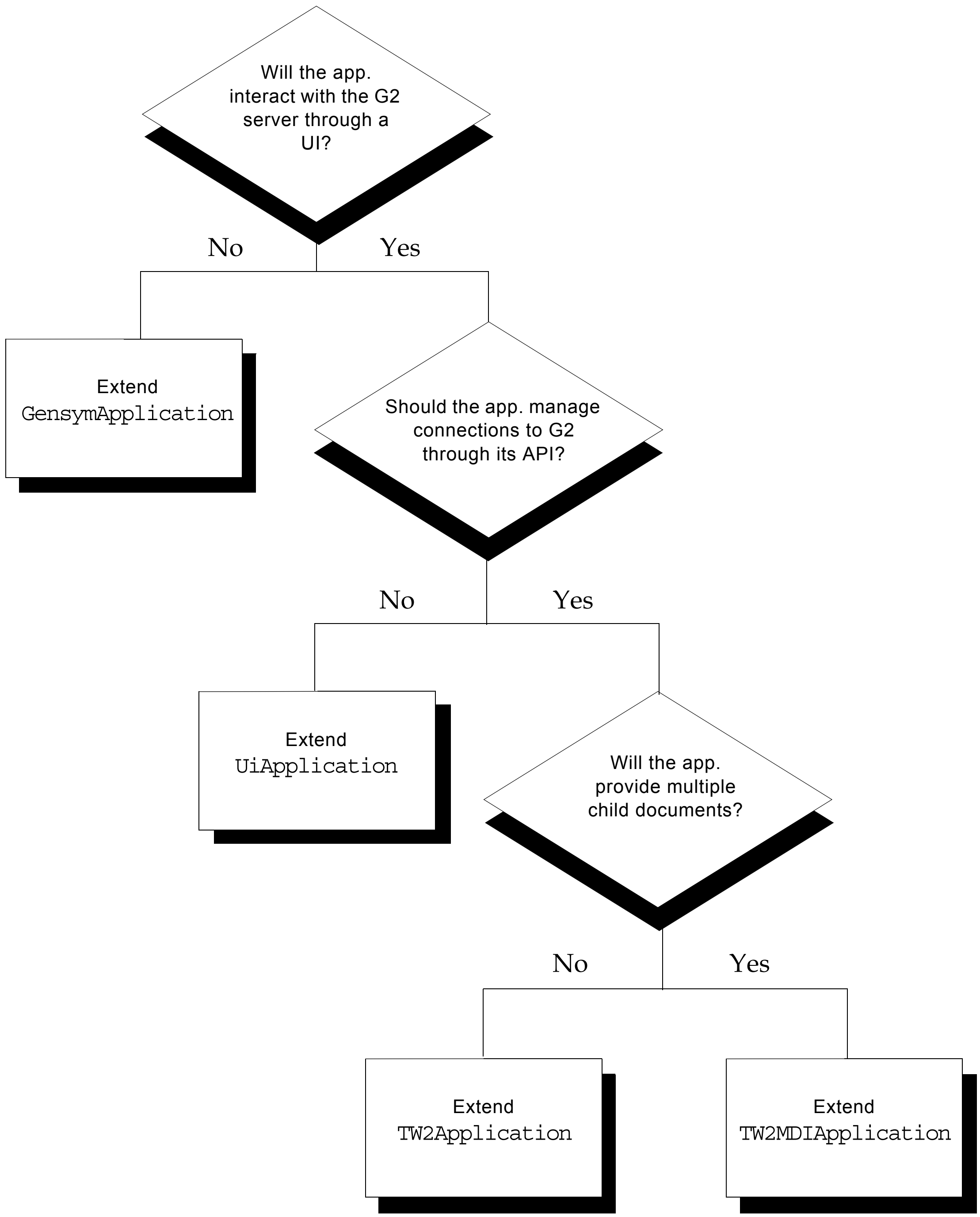
To create this type of application...	Extend this class...
SDI application that handles connections to G2	`com.gensym.shell.util.TW2Application`
MDI application that handle connections to G2	`com.gensym.shell.util.TW2MDIApplication`

For specific information on creating each type of applications, see:

- “Creating Single Document Interface Applications” on page 247.
- “Creating Multiple Document Interface Applications” on page 251.

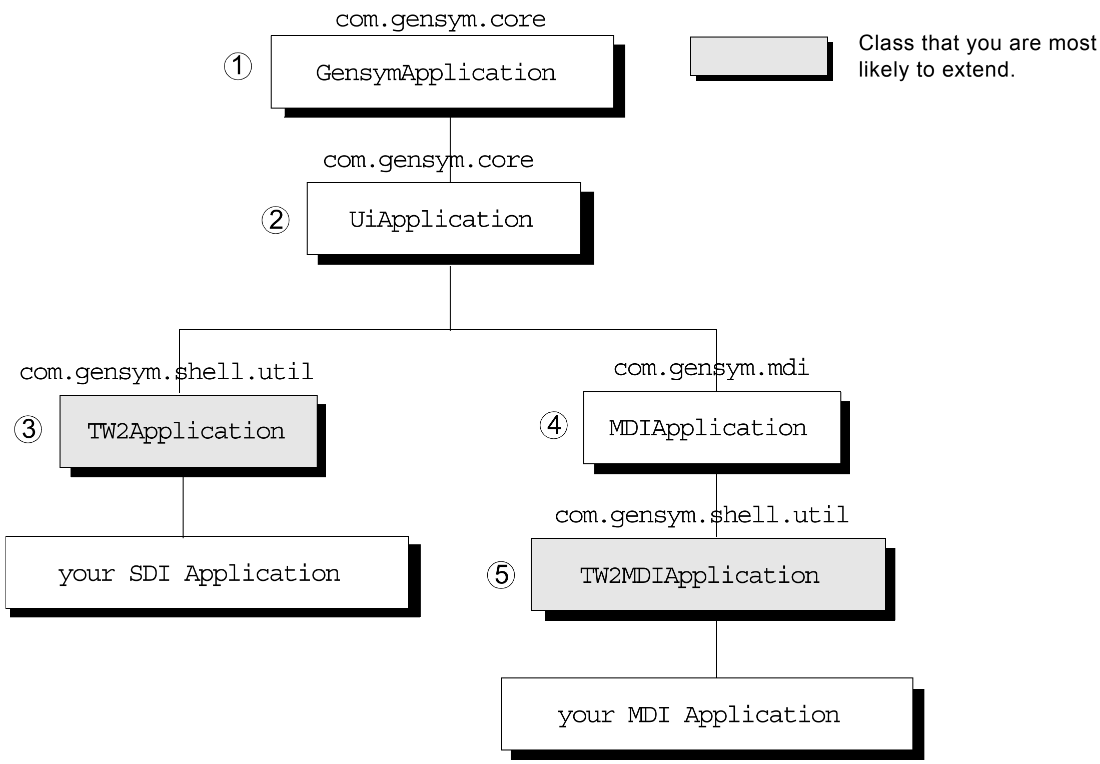
Decision Tree to Determine Which Class to Extend

Use the following decision tree to determine which application foundation class you should extend:



Application Foundation Classes

The following diagram shows the inheritance structure of an SDI or MDI application that you might create:



1	Parses and handles command-line arguments that are the root of any G2 client application.
2	Parses command-line arguments that determine the title and geometry of the frame.
3	Parses command-line arguments for connecting to G2 and making a secure login.
4	Provides a place holder for creating MDI applications.
5	Parses command-line arguments for connecting to G2 and making a secure login.

The following sections explain the features, behavior, and key methods of each of these classes.

GensymApplication

Features

`GensymApplication` is a G2 JavaLink class, which all G2 client applications should extend, either directly or through one of its subclasses. This class provides a generic G2 client application that interacts with the G2 server through its data.

Behavior

`GensymApplication` provides this behavior:

- Parses and handles these command-line arguments, which deal with internationalization and debugging:

  ```
  -language language-code
  -country country-code
  -variant variant-code
  -development
  ```

Argument	Description
language-code	Lowercase two-letter ISO-639 code.
country-code	Uppercase two-letter ISO-3166 code.
variant-code	Vendor- and browser-specific code.

  By specifying these command-line arguments, you override the value returned by calling `getDefault()` on a `java.util.Locale`. For details, see `java.util.Locale`.

- Initializes the application by parsing the `.com.gensym.properties` file.
- Initializes system properties that need to be set for some classes to function, including paths to the `URLStreamHandlers` and `ContentHandlers`.

Note Unlike the subclasses of `GensymApplication`, which just parse their command-line arguments, `GensymApplication` both parses and handles its command-line arguments as part of the application.

Methods

`GensymApplication` supports these two static methods:

- `initialize(String commandLine[])` — Parses and handles command-line arguments, which the `GensymApplication` constructor calls, and which any application can call as a static method to parse command line arguments if it does not extend `GensymApplication`.
- `getApplication()` — Returns a handle to your application, which any application can call as a static method, assuming the application has been created.

UiApplication

Features

`UiApplication` creates a generic application that interacts visually with G2 through some kind of user interface.

`UiApplication` extends `GensymApplication`.

Behavior

`UiApplication` parses these command-line arguments that deal with the application frame:

```
-title title
-geometry widthXheight[+x+y][-x-y]
```

Argument	Description
title	The title of the application's window as a `java.lang.String`.
widthXheight[*+x+y*][*-x-y*]	The width and height in pixels of the application window, separated by an "x", with optional x and y offsets. Positive values represent offsets from the top-left corner, and negative values represent offsets from the bottom-right corner.

Methods

`UiApplication` provides a number of useful methods, including:

- `setCurrentFrame(Frame frame)` — Sets the current frame to any `java.awt.Frame`.
- `getCurrentFrame()` — Returns the frame that `setCurrentFrame` sets.
- `getTitleInformation()` — Returns the value of the `-title` command-line argument as a `java.lang.String`.
- `getGeometryInformation()` — Returns the value of the `-geometry` command-line argument as a `java.lang.String`.
- `parseBounds(String optn)` — Parses the `widthXheight` argument of the `-geometry` command-line argument, given as a `java.lang.String`, and returns a `java.awt.Rectangle`.
- `setBoundsForFrame(Frame frame, String geometry)` — Sets the dimensions of the `java.awt.Frame`, using the `geometry` argument, which is the return value of the `getGeometryInformation` method.
- `initialize(String commandLine[])` — Parses and handles command-line arguments, which the `UiApplication` constructor calls, and which any application can also call statically to parse command-line arguments.

TW2Application

Features

`TW2Application` creates an SDI application that manages:

- A single application frame.
- Single and multiple connections to G2.

`TW2Application` extends `UIApplication`.

For details on using this class, see "Creating Single Document Interface Applications" on page 247.

Behavior

`TW2Application` parses command-line arguments that deal with:

- Connecting to G2:

```
-url url-location
-host host-name
-port port-number
```

- Logging on to a secure G2:

```
-userName login-name
-userMode user-mode
-password password
```

Command-Line Argument	Description
url-location	A URL to a middle tier when running TW2 Toolkit in 3-tier mode. For more information, see Chapter 8 “Using a Middle-Tier Server” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.
host-name	The name of a computer on which the G2 server is running, as a `java.lang.String`.
port-number	The port on which the G2 server is running, as a `java.lang.String`.
`login-name`	The login name of a user on the network, as a `java.lang.String`.
`user-mode`	The name of an existing G2 user mode, as a `java.lang.String`.
`password`	The user’s password for logging on to a secure G2, as a `java.lang.String`.

Methods

`TW2Application` provides a number of useful methods, including:

- `getConnection()` — Returns the `com.gensym.ntw.TwAccess` that is the current connection in an application that connects to a single G2.
- `setConnection(TwAccess connection)` — Specifies the behavior of an application that connects to a single G2 when it connects.
- `getConnectionManager()` — Returns the `com.gensym.shell.util.ConnectionManager` for an application that supports multiple connections to G2.
- `getG2ConnectionInformation()` — Returns a `com.gensym.ntw.TwConnectionInfo` that contains the host, port, and URL obtained from parsing the command line. You can pass the `TwConnectionInfo` as the argument to the `com.gensym.ntw.TwGateway.openConnection` static method to connect to G2.

For details, see Chapter 5 “Using Connection Information Objects” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

- `getLoginRequest()` — Returns a `com.gensym.ntw.LoginRequest` that contains the user name, user mode, and password obtained from parsing the command line. You can pass the `LoginRequest` as the argument to the `login` method on a `TwGateway` to make a secure login to G2.

  For details, see Chapter 7 “Establishing a Login Session” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

- `initialize(String commandLine[])` — Parses and handles command-line arguments, which the `TW2Application` constructor calls, but which any application can also call statically.

MDIApplication

`MDIApplication` extends `UiApplication`. Currently, the `MDIApplication` class defines no methods and is simply a place-holder for creating generic multiple document interface applications.

TW2MDIApplication

Features

`TW2MDIApplication` creates a MDI application that manages:

- An `MDIFrame`.
- Single and multiple connections to G2.

`TW2MDIApplication` extends `MDIApplication`.

For details on using this class, see “Creating Multiple Document Interface Applications” on page 251.

Behavior

`TW2MDIApplication` parses command-line arguments that deal with:

- Connecting to G2:

  ```
  -url url-location
  -host host-name
  -port port-number
  ```

- Logging into a secure G2:

  ```
  -userName login-name
  -userMode user-mode
  -password password
  ```

For a description of these command line arguments, see “TW2Application” on page 230.

Methods

`TW2MDIApplication` supports the same methods that `TW2Application` supports. For details, see “TW2Application” on page 230.

Summary of Application Foundation Class Features

This table summarizes the features your application supports when you extend each of the application foundation classes:

Application Foundation Class	SDI	MDI	Single Connection	Multiple Connection	Visible Frame
`GensymApplication`					
`UiApplication`					✓
`TW2Application`	✓		✓	✓	✓
`MDIApplication`		✓			✓
`TW2MDIApplication`		✓	✓	✓	✓

Creating Telewindows2 Toolkit Applications

This section provides a summary of the required and optional features of subclasses of `TW2Application` and `TW2MDIApplication`.

The specific steps required to implement each feature are described in these sections:

- “Creating Single Document Interface Applications” on page 247.
- “Creating Multiple Document Interface Applications” on page 251.
- “Creating and Managing Connections to G2” on page 236.

The summary sections that follow provide specific references within each of these sections for details on implementing each feature.

Required Features of SDI and MDI Applications

Subclasses of `TW2Application` and `TW2MDIApplication` must implement the following required features. The implementation of these features depends on the type of application.

Creating and Managing the Application Frame

Subclasses of `TW2Application` and `TW2MDIApplication` are responsible for:

- Creating the frame.
- Setting the current frame by calling `setFrame` on the application.
- Making the frame visible.
- Adding UI controls to the frame, for example, menus and toolbars.

To create and manage...	See...
SDI application frames	"Creating and Setting the Frame in an SDI Application" on page 249.
MDI application frames	"Creating and Setting the Frame in an MDI Application" on page 252.

For information on adding UI controls to an application frame, see Chapter 5, "Creating Menus and Toolbars" on page 113.

Creating and Managing Connections to G2

Subclasses of `TW2Application` and `TW2MDIApplication` are responsible for creating and managing connections to G2.

To create and manage connections, use the following classes:

To...	Use this class...
Create single connections to G2	`com.gensym.ntw.TwGateway`
Create and manage multiple connections to G2	`com.gensym.shell.util.ConnectionManager`

For information on creating and managing single and multiple connections, see "Creating and Managing Connections to G2" on page 236.

Implementing Abstract Methods

Subclasses of `TW2Application` and `TW2MDIApplication` must implement the following abstract methods:

- `getConnection` — Gets the connection in an application that supports single connections to G2.
- `getConnectionManager` — Gets the `ConnectionManager` in an application that supports multiple connections to G2.
- `setConnection` — Determines the behavior of an application that supports single connections to G2 when the connection opens or closes.

The implementation of these methods depends on whether your application supports single or multiple connections.

For details, see “Implementing Abstract Methods to Manage Connections” on page 244.

Optional Features of SDI and MDI Applications

Subclasses of `TW2Application` and `TW2MDIApplication` typically implement a number of optional features, whose implementation is the same for either type of application. These features include:

- Parsing and handling command line arguments that provide the application frame and connection information.
- Implementing event listeners.
- Setting the look and feel of the Java UI classes.
- Localizing application text.

For examples of these optional features, see the Chapter 12, “Understanding the Telewindows2 Toolkit Shell,”

Optional Feature Specific to SDI and MDI Applications

Subclasses of `TW2Application` and `TW2MDIApplication` can be listeners for programmatic show and hide KB workspace events in G2 by using adapter classes in the `com.gensym.shell.util` package. The implement of this optional feature depends on the type of application.

For information on implementing this feature in...	See...
SDI applications	"Listening for Programmatic Show and Hide KB Workspace Events in SDI Applications" on page 250.
MDI applications	• "Listening for Programmatic Show and Hide KB Workspace Events in an MDI Application" on page 254. • "Registering Workspace Document Factories" on page 255.

Creating and Managing Connections to G2

The most fundamental feature of any G2 client application is its connections to G2, for it is through the G2 connection that the client has access to all G2 server data.

Depending on the type of connection you want to support, your application uses different connection classes to create and manage G2 connections, and listen for connection events.

To determine which connection and listener classes your application should use, you need to answer the question in the following heading.

Will the Application Support Single or Multiple Connections to G2?

Regardless of whether you are extending `TW2Application` or `TW2MDIApplication`, you can create one of the following types of applications:

- **Single connection application**, which is an application that connects to a single G2 server.
- **Multiple connection application**, which is an application that allows multiple connections to different G2 servers.

The following sections describe how to create and manage multiple connections to G2 by using a `ConnectionManager`, which includes:

- Creating a connection manager.
- Opening connections.

- Getting connection and login information.
- Getting and setting the current connection.
- Listening for changes in the current connection context.
- Implementing abstract methods that manage connections.

If your application only needs to create connection to a single G2, use a `com.gensym.ntw.TwGateway`. For details, see Chapter 6 “Using TwGateway” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

Creating a ConnectionManager

The `ConnectionManager` is responsible for:

- Opening connections.
- Keeping track of the current connection.
- Setting a new connection to be the current connection.
- Maintaining a list of all open connections.
- Notifying registered `ContextChangedListeners` when the current connection context changes.

Because so many features of a multiple connection application depend on the G2 connection, you typically create a `ConnectionManager` in the application’s constructor.

To create a `ConnectionManager`:

➔ Create a single instance of this class in the constructor for your application:

```
com.gensym.shell.util.ConnectionManager
```

For example:

```
ConnectionManager connectionMgr = new ConnectionManager()
```

Opening a Connection through a ConnectionManager

When you open a connection through a `ConnectionManager`, the manager:

- Opens the connection.
- Sets the new connection to be the current connection.
- Notifies registered listeners of `ContextChangedEvents`.
- Handles exceptions, if the connection fails.

To open a connection through a `ConnectionManager`:

➔ Call one of the following methods on `ConnectionManager`:

- `openConnection(String url, String host, String port)`, where the arguments are all instances of a `java.lang.String`.
- `openConnection(TwConnectionInfo connectionInfo)`, where `connectionInfo` is an instance of a `com.gensym.ntw.TwConnectionInfo`.

Both methods return an implementation of this interface, such as a `TwGateway`:

```
com.gensym.ntw.TwAccess
```

For details on these core classes, see these chapters in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*:

- Chapter 5, "Using Connection Information Objects."
- Chapter 6, "Using TwGateway."

Getting Connection and Login Information

Subclasses of `TW2Application` and `TW2MDIApplication` support methods for parsing the following information from the command line:

- Connection information, which you can pass as the argument to the `openConnection` method on a `ConnectionManager` to open a connection.
- Login information, which you can pass as the argument to the `login` method on an implementation of `com.gensym.ntw.TwAccess`, such as `TwGateway`, to log on to a secure G2.

To get connection information from the command line:

➔ Call this method on a subclass of `TW2Application` or `TW2MDIApplication`:

```
getG2ConnectionInformation()
```

This method returns an instance of this class, which holds the value of the `-host`, `-port`, and `-url` command-line arguments:

```
com.gensym.ntw.TwConnectionInfo
```

For information on this core class, see Chapter 5, "Using Connection Information Objects" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

To get login request from the command line:

➔ Call this method on a subclass of `TW2Application` or `TW2MDIApplication`:

```
getLoginRequest()
```

This method returns an instance of this class, which holds the value of the -userName, -userMode, and -password command-line arguments:

```
com.gensym.ntw.LoginRequest
```

For information on this core class, see Chapter 7, "Establishing a Login Session" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

Example

Using Command-Line Arguments to Open a G2 Connection

Subclasses of TW2Application or TW2MDIApplication supports command-line arguments for specifying the host, port, and URL, which your application can use to make a connection to G2.

Your application can also provide other command-line arguments for connecting to G2. For example, you might define a single command-line argument to take the host and port of the G2 to which to connect, for example, -g2 localhost 1111. You would use this command-line argument to create the same TwConnectionInfo that the getG2ConnectionInformation method returns.

For example, this code fragment might appear in the main method of a subclass of TW2Application or TW2MDIApplication that supports multiple connections to G2, where application is your application. The code opens a connection to a secure G2 by parsing command-line arguments:

```
try {
    ConnectionManager connectionMgr =
        application.getConnectionManager();
    TwConnectionInfo connectionInfo =
getG2ConnectionInformation();
    if (connectionInfo != null) {
        TwAccess cxn = connectionMgr.
openConnection(connectionInfo);
    LoginRequest loginRequest = getLoginRequest();
    if (loginRequest != null){
        if (cxn != null)
            cxn.login(loginRequest);
        }
    }
}

catch (G2AccessException e) {
    e.printStackTrace();
}
```

Getting and Setting the Current Connection

You get the current connection from a `ConnectionManager` for numerous reasons, including to:

- Make a login request to a secure G2.
- Get and set the user mode.
- Get and set the G2 run state.
- Close the current connection.
- Get unique named items from G2, such as a named workspace or a named item.
- Make RPC calls.
- Set the availability of a command based on the existence of a connection.
- Get the `com.gensym.util.ClassManager`, which manages G2 class definitions on the client.
- Get the `com.gensym.dlgruntime.DialogManager`, which manages G2 item properties dialogs in the client.

You can also get a list of all open connections from a `ConnectionManager`.

As mentioned earlier, when you open a connection through a `ConnectionManager`, the manager sets the connection as the current connection automatically. Thus, the only time you need to set the current connection is in a single connection application, which must switch the current connection explicitly.

Getting the Current Connection

To get the current connection:

➔ Call this method on a `ConnectionManager`:

```
getCurrentConnection()
```

This method typically returns an implementation of this interface, such as a `TwGateway`:

```
com.gensym.ntw.TwAccess
```

For information on this core class, see Chapter 6, “Using TwGateway” in the *Telewindows2 Toolkit Java Developer’s Guide: Components and Core Classes*.

For example, to make a login request, you must first get the current connection from the ConnectionManager:

```
LoginRequest loginRequest = getLoginRequest();
if (loginRequest != null){
    TwAccess cxn = connectionMgr.getCurrentConnection();
    if (cxn != null)
        cxn.login(loginRequest);
}
```

Getting a List of Open Connections

You typically need to get a list of open connections before you exit the application so you can explicitly close each open connection.

To get a list of all open connections:

➔ Call this method on a ConnectionManager:

```
getCurrentConnections()
```

This method returns an array of objects, each of which is an implementation of this interface, typically a TwGateway:

```
com.gensym.ntw.TwAccess
```

For information on this core class, see Chapter 6, "Using TwGateway" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

For example, this method exits a multiple connection application by getting, then closing each element in the array of open connections:

```
private void exitApp(ConnectionManager connectionManager) {
    TwAccess[] cxns = connectionManager.getOpenConnections();
    for (int i=0; i<cxns.length; i++)
        cxns[i].closeConnection();
    System.exit(0);
}
```

Setting the Current Connection

When you open a connection through a ConnectionManager, the manager automatically sets the open connection to be the current connection. Thus, when you open a new connection, you do not need to be concerned with setting the current connection.

However, if your application supports switching the connection, for example, through a dialog, you must set the current connection to the selected connection explicitly.

To set the current connection:

➔ Call this method on a `ConnectionManager`:

```
setCurrentConnection(TwAccess connection)
```

The argument to the method is an instance of a class that implements this interface, typically a `TwGateway`:

```
com.gensym.ntw.TwAccess
```

For information on this core class, see Chapter 6, "Using TwGateway" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

For example, the following code fragment gets the selected connection for the current command key from `connectionTable`. This table maps connection names to open connections. The code then sets the current connection to the selected connection.

```
private java.util.HashTable connectionTable;

TwAccess connection =
   (TwAccess)connectionTable.get(cmdKey);
   if (connection != null){
      connectionMgr.setCurrentConnection(connection);
```

Listening for Changes in the Current Connection Context

Various classes in your application might need to be notified when the current connection context changes, that is, when a new connection becomes the current connection, as maintained by the `ConnectionManager`.

For example, a command that makes an RPC call to G2 should only be available if a current connection exists.

The `ConnectionManager` delivers a `ContextChangedEvent` to all registered `ContextChangedListeners` by calling their `currentConnectionChanged` method when the current connection context changes.

If the last connection in the list of open connections closes, the event still occurs, but the current connection is `null`.

You get the current connection by calling `getConnection` on the `ContextChangedEvent`.

To listen for changes in the current connection context:

1 Implement this interface:

```
com.gensym.shell.util.ContextChangedListener
```

2 Register the class that needs to receive notification of `ContextChangedEvents` as a `ContextChangedListener` by calling the appropriate add method on a `ConnectionManager`.

For example, this code fragment would appear in the constructor of a class that implements the `ContextChangedListener` interface:

```
private ConnectionManager connectionMgr;

connectionMgr.addContextChangedListener(this);
```

Example

Creating a Command that Listens for ContextChangedEvents

The following code fragments implements a command that displays a named workspace. The command:

- Implements the `ContextChangedListener` interface.
- Adds itself as a listener.
- Implements the `currentConnectionChanged` abstract method to set the command's availability when the current connection context changes.

For information on the arguments to `CommandInformation` and general information on creating commands, see "Creating Commands" on page 131.

This code fragment shows the definition of the class that implements the `ContextChangedListener` interface:

```
public ViewCommands extends AbstractCommand
    implements ContextChangedListener {
    //Additional code
}
```

Here is the constructor for the command, which adds itself as a `ContextChangedListener`:

```
public ViewCommands(MDIFrame frame,
                    ConnectionManager connectionManager) {
   super(new CommandInformation[]{
      new CommandInformation(SCHEMATIC, true,
                             shortResource, longResource,
                             null, null, null, false)}),
   this.frame = frame;
   this.connectionMgr = connectionManager;
   connectionMgr.addContextChangedListener(this);
}
```

The following method implements the abstract method of the `ContextChangedListener` interface. You call `getConnection` on the `ContextChangedEvent` argument to get the current connection. The method makes the command unavailable when no current connection exists.

```
public void currentConnectionChanged(ContextChangedEvent e){
   TwAccess context = e.getConnection();
   boolean available = true;
   if (context == null)
      available = false;
   setAvailable(SCHEMATIC, available);
}
```

Implementing Abstract Methods to Manage Connections

Subclasses of `TW2Application` and `TW2MDIApplication` require that you implement three abstract methods that allow the application to manage single and multiple connections:

- `getConnection()` — Gets the current connection in single connection applications.
- `getConnectionManager()` — Gets the `ConnectionManager` in multiple connection applications.
- `setConnection(connection)` — Specifies the behavior of single connection applications when the current connection is set.

The implementation of these abstract methods depends on the type of application, as this table describes:

	Single Connection Applications	Multiple Connection Applications
getConnection	Returns the current connection	Returns `null`
getConnectionManager	Returns `null`	Returns the `ConnectionManager`
setConnection	Implements the behavior when a connection opens or closes	Empty implementation

Note Although you must implement the `setConnection` abstract method in a multiple connection application, it should not return anything.

Implementing the Get Methods in a Single Connection Application

The following examples show implementations of `getConnection` and `getConnectionManager` for single connection applications:

```
private TwAccess connection;

public TwAccess getConnection() {
   return connection;
}

public ConnectionManager getConnectionManager() {
   return null;
}
```

Implementing the Get Methods in a Multiple Connection Application

The following examples show implementations of `getConnection` and `getConnectionManager` for multiple connection applications:

```
private ConnectionManager connectionManager;

public ConnectionManager getConnectionManager() {
   return connectionManager;
}

public TwAccess getConnection () {
   return null;
}
```

Setting the Connection

The `setConnection` method is an abstract method that your single connection `TW2Application` or `TW2MDIApplication` must implement to track the current connection. When a connection opens or closes, the application needs to handle certain events, which it typically does in the `setConnection` method. The `setConnection` method also determines the connection that the `getConnection` method returns.

For example, you might define the `setConnection` method to update the connection and user mode in the toolbar when the connection changes.

The following implementation of the `setConnection` method performs these tasks:

- Tests to see if a connection exists.
- If a connection exists:
  - Adds a `com.gensym.wksp.MultipleWorkspacePanel` as a `com.gensym.ntw.WorkspaceShowingListener`.

    For details on these classes, see these chapters in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*:

    - Chapter 13, "Using Workspace View Components."
    - Chapter 6, "Using TwGateway."
  - Updates the connection and user mode in the `com.gensym.shell.util.HostPortPanel` and `UserModePanel`.

    For details on these classes, see Chapter 10, "Using Shell Dialogs and UI Controls" on page 259.
- If no connection exists:
  - Removes all workspaces.
  - Removes the listener from the `com.gensym.ntw.TwGateway`.
  - Updates the connection and user mode in the toolbar panel.
- Sets the new connection as the current connection.
- Notifies listeners that the connection has been updated.

Here is an implementation of the `setConnection` method:

```
private MultipleWorkspacePanel multiWkspPanel;
private HostPortPanel hostPortPanel;
private UserModePanel userModePanel;
private TwAccess connection;

public void setConnection (TwAccess newCxn) {
   boolean connected = (newCxn != null);
```

```
        //If a connection exists
        if (connected) {
            try {
                newCxn.addWorkspaceShowingListener (multiWkspPanel);
                hostPortPanel.setConnection(null);
                userModePanel.setConnection(null);
            } catch (G2AccessException e) {
                new WarningDialog (null, "Error Setting Connection",
                    true, e.toString (), null).setVisible (true);
                e.printStackTrace();
            }
        //If a connection does not exist
        } else {
            try {
                KbWorkspace[] showingWorkspaces =
                    multiWkspPanel.getWorkspaces ();
            for (int i=0; i<showingWorkspaces.length; i++)
                multiWkspPanel.removeWorkspace (showingWorkspaces[i]);
                Rectangle frameRect = getCurrentFrame().getBounds ();
                connection.removeWorkspaceShowingListener (multiWkspPanel);
                hostPortPanel.setConnection ((TwConnection)newCxn);
                userModePanel.setConnection ((TwConnection)newCxn);
            } catch (G2AccessException e) {
                new WarningDialog (null, "Error Disconnecting
                    Connection", true, gae.toString (), null).
                    setVisible (true);
                e.printStackTrace();
            }
        }
    }
```

Creating Single Document Interface Applications

An SDI application contains a single frame that displays a view into the G2 server's data, typically a workspace view.

For example, your SDI application might display a single KB workspace by adding a `com.gensym.wskp.ScalableWorkspaceView` component to the frame. Alternatively, your application could add one of the multiple workspace view components to support switching between multiple KB workspaces from within a single application frame.

For information on workspace view components, see Part III "Viewing Workspaces" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

This section summarizes how to implement the:

- Required features of an SDI application.
- Optional features specific to SDI applications.

For information on additional optional features, see "Optional Features of SDI and MDI Applications" on page 235.

For demonstrations that illustrate SDI applications, see the source code for these classes:

```
com.gensym.demos.wksppanel.BrowserApplication
com.gensym.demos.singlecxnsdiapp.BrowsesrApplication
com.gensym.demos.multiplecxnsdiapp.WorkspaceBrowserApp
```

The following steps summarize how to implement an SDI application and provide references to other sections for details.

To implement an SDI application:

1 Create an application that extends this application foundation class:

```
com.gensym.shell.util.TW2Application
```

2 Create and set the application frame, and make it visible.

For details, see "Creating and Setting the Frame in an SDI Application" on page 249.

3 Create and manage single or multiple connections to G2.

For details, see "Creating and Managing Connections to G2" on page 236.

4 Implement these abstract methods on `TW2Application`:

```
getConnection()
getConnectionManager()
setConnection()
```

For details, see "Implementing Abstract Methods to Manage Connections" on page 244.

5 Make the application be a listener for `WorkspaceShowingEvents`, as needed.

For details, see "Listening for Programmatic Show and Hide KB Workspace Events in SDI Applications" on page 250.

Creating and Setting the Frame in an SDI Application

To create the frame of an SDI application, you typically create an instance of one of these classes:

- `java.awt.Frame`
- `javax.swing.JFrame`

Once you create the frame, you set it as the current frame, then get the frame and make it visible.

`TW2Application` inherits from `UiApplication` the methods that support getting and setting the current frame. For details, see "UiApplication" on page 229.

To create and set the application frame:

1. In the application's constructor, call the constructor for the superior class, passing the command-line arguments as its arguments.
2. To parse the title from the command line, call this method:

   ```
   getTitleInformation()
   ```

3. Create an instance of the frame.
4. Call this method on the application to set the frame as the current frame:

   ```
   setCurrentFrame(Frame frame)
   ```

For example, this constructor for a subclass of `TW2Application` creates an instance of a `javax.swing.JFrame`, passing the title from the command line as its argument:

```
public SDIApplication (String[] cmdLineArgs) {
   super (cmdLineArgs);
   JFrame jf;
   String title = getTitleInformation ();
   setCurrentFrame (jf = new JFrame
      (title != null ? title : "SDI Application"));
}
```

To make the frame visible:

1. In the `main` method, create an instance of your application class.
2. Get the current frame from the application by calling this method:

   ```
   getCurrentFrame()
   ```

3. Make the frame visible by calling this method on the frame:

   ```
   setVisible(true)
   ```

For example, this code fragment show the part of the `main` method that gets the current frame from the application and makes it visible:

```
public static void main (String[] args) {
   SDIApplication app = new SDIApplication (args);
   app.getCurrentFrame().setVisible (true);

   // Additional code

}
```

Listening for Programmatic Show and Hide KB Workspace Events in SDI Applications

Your application can listen for programmatic show and hide KB workspace events in G2 by implementing the `com.gensym.ntw.WorkspaceShowingListener` interface. This interface provides abstract methods that determine the behavior of registered listeners when G2 programmatically shows or hides a KB workspace.

For details, see Chapter 6, "Using TwGateway" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

TW2 Toolkit provides the following adapter class as a default implementation of `WorkspaceShowingListener` for SDI applications:

```
com.gensym.shell.util.TW2WorkspaceShowingAdapter
```

This adapter class performs these tasks when the SDI application receives notification of a `WorkspaceShowingEvent`:

When G2 programmatically...	Your SDI application...
Shows a KB workspace	Adds a `com.gensym.wksp.ScalableWorkspaceView` to the application frame.
Hides a KB workspace	Removes a `com.gensym.wksp.ScalableWorkspaceView` from the application frame.

For information on workspace views, see Part III "Viewing Workspaces" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

TW2WorkspaceShowingAdapter provides two constructors, which take different arguments, depending on whether the SDI application supports single or multiple connections to G2:

Use this constructor...	To support...
TW2WorkspaceShowingAdapter(TwAccess connection)	Single connections
TW2WorkspaceShowingAdapter (ConnectionManager connectionManager)	Multiple connections

To add and remove workspace views based on WorkspaceShowingEvents:

➔ In the main method for your SDI application, create an instance of this adapter, using the constructor that supports your type of connection:

```
com.gensym.shell.util.TW2WorkspaceShowingAdapter
```

For example, the following code fragment appears in the main method of a single connection SDI application that listens for WorkspaceShowingEvents. The constructor takes a single connection as its argument, which it obtains from the application.

```
private TWApplication app;
private TwAccess connection;

TW2WorkspaceShowingAdapter wkspShowingListener =
    new TW2WorkspaceShowingAdapter(app.connection);
```

Creating Multiple Document Interface Applications

An MDI application contains an MDIFrame which consists of one or more MDIDocuments, each of which contains a workspace view, or some other view into the G2 server's data.

This section summarizes how to implement the:

- Required features of an MDI application.
- Optional features specific to MDI applications.

For information on additional optional features, see "Optional Features of SDI and MDI Applications" on page 235.

For information on the containers you can use to create MDI applications, see Chapter 7, "Creating Multiple Document Interface Containers" on page 187.

For demonstrations that illustrate MDI applications, see the source code for these classes:

- `com.gensym.demos.singlecxnmdiapp.SingleConnectionApplication`
- `com.gensym.demos.multiplecxnmdiapp.Shell`
- `com.gensym.shell.Shell`

For a complete code walk-through of the `Shell.java` source code, see Chapter 12, "Understanding the Telewindows2 Toolkit Shell" on page 301.

The following steps summarize how to implement an MDI application and provide references to other sections for details.

To create an MDI application:

1 Create an application that extends this application foundation class:

```
com.gensym.shell.util.TW2MDIApplication
```

2 Create and set the application frame, and make it visible.

For details, see "Creating and Setting the Frame in an MDI Application" on page 252.

3 Create and manage single or multiple connections to G2.

For details, see "Creating and Managing Connections to G2" on page 236.

4 Implement these abstract methods on `TW2MDIApplication`:

```
getConnection()
getConnectionManager()
setConnection()
```

For details, see "Implementing Abstract Methods to Manage Connections" on page 244.

5 Make the application be a listener for `WorkspaceShowingEvents`, as needed.

For details, see "Listening for Programmatic Show and Hide KB Workspace Events in an MDI Application" on page 254.

6 Register implementations of the `WorkspaceDocumentFactory` interface.

For details, see "Registering Workspace Document Factories" on page 255.

Creating and Setting the Frame in an MDI Application

You create the application frame by creating an instance of this class:

```
com.gensym.mdi.MDIFrame
```

For details on this class, see "Creating and Managing MDI Frames" on page 193.

Once you create the frame, you set it as the current frame, then get the frame and make it visible.

`TW2MDIApplication` inherits from `UiApplication` the methods that support getting and setting the current frame. For details, see “UiApplication” on page 229.

To create and set the application frame:

1. In the application’s constructor, call the constructor for the superior class, passing the command-line arguments as its argument.
2. To parse the title from the command line, call this method:

   ```
   getTitleInformation()
   ```

3. Create an instance of an `MDIFrame`.
4. Call this method on the application to set the `MDIFrame` as the current frame:

   ```
   setCurrentFrame(Frame frame)
   ```

For example, these code fragments appear in the constructor for a subclass of `TW2MDIApplication` to create an instance of an `MDIFrame` and set it as the current frame. The frame uses a localized text string for its title.

```
private com.gensym.message.Resource i18nUI;

MDIFrame mdiFrame = createFrame(i18nUI.getString("MDIAppTitle"));
setCurrentFrame(mdiFrame);
```

To make the frame visible:

1. In the `main` method, create an instance of your application class.
2. Get the current frame from the application by calling this method:

   ```
   getCurrentFrame()
   ```

3. Make the frame visible by calling this method on the frame:

   ```
   setVisible(true)
   ```

For example, this code fragment shows the part of the `main` method that gets the current frame from the application and makes it visible:

```
public static void main (String[] args) {
   MDIApplication app = new MDIApplication (args);
   app.getCurrentFrame().setVisible (true);

      // Additional code
   }
```

Listing for Programmatic Show and Hide KB Workspace Events in an MDI Application

Your application can listen for programmatic show and hide KB workspace events in G2 by implementing the `com.gensym.ntw.WorkspaceShowingListener` interface. This interface provides abstract methods that determine the listener's behavior when G2 programmatically shows or hides a workspace.

For details, see Chapter 6, "Using TwGateway" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

TW2 Toolkit provides the following adapter class as a default implementation of `WorkspaceShowingListener` for MDI applications:

```
com.gensym.shell.util.TW2MDIWorkspaceShowingAdapter
```

This adapter class performs these tasks when the MDI application receives notification of a `WorkspaceShowingEvent`:

When G2 programmatically...	The MDI application...
Shows a KB workspace	Adds a `com.gensym.wksp.ScalableWorkspaceView` to a `com.gensym.shell.util.WorkspaceDocument` and displays the document in the frame.
Hides a KB workspace	Removes a `com.gensym.shell.util.WorkspaceDocument` with its `com.gensym.wksp.ScalableWorkspaceView` from the frame.

By default, the adapter uses a `com.gensym.shell.util.DefaultWorkspaceDocumentFactoryImpl` to generate a `WorkspaceDocument` whenever G2 programmatically shows a KB workspace. If your MDI application needs to create instances of a `WorkspaceDocument` subclass in which to display a workspace view, you must also register the factory used to generate the workspace document, as described in "Registering Workspace Document Factories" on page 255.

For details on workspace views, workspace documents, and workspace document factories, see:

- Part III "Viewing Workspaces" in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.
- Chapter 8, "Using Telewindows2 Toolkit MDI Documents" on page 207.

TW2MDIWorkspaceShowingAdapter provides two constructors, which take different arguments, depending on whether the MDI application supports single or multiple connections to G2:

Use this constructor...	To support...
TW2MDIWorkspaceShowingAdapter(TwAccess connection)	Single connections
TW2MDIWorkspaceShowingAdapter (ConnectionManager connectionManager)	Multiple connections

To add and remove WorkspaceDocuments based on WorkspaceShowingEvents:

➔ In the main method for your MDI application, create an instance of this class, using the constructor that supports your type of connection:

```
com.gensym.shell.util.TW2MDIWorkspaceShowingAdapter
```

For example, this code fragment appears in the main method of a multiple connection MDI application that listens for WorkspaceShowingEvents. The constructor takes a ConnectionManager as its argument, which it obtains from the application.

```
private Tw2MDIApplication app;
private ConnectionManager connectionManager;

TW2MDIWorkspaceShowingAdapter wkspShowingListener =
   new TW2MDIWorkspaceShowingAdapter(app.connectionManager);
```

Registering Workspace Document Factories

Your MDI application typically creates instances of a com.gensym.shell.util. WorkspaceDocument. For example, the com.gensym.shell.Shell class uses a com.gensym.shell.commands.WorkspaceCommands to create workspace documents when the user chooses a named workspace. Similarly, the Shell class uses a com.gensym.shell.util.TW2MDIWorkspaceShowingAdapter class to create workspace documents when G2 programmatically shows a KB workspace.

For information on these classes, see:

- "Listening for Programmatic Show and Hide KB Workspace Events in an MDI Application" on page 254.
- "WorkspaceCommands" on page 293.
- Chapter 12, "Understanding the Telewindows2 Toolkit Shell" on page 301.

By default, WorkspaceCommands and TW2MDIWorkspaceShowingAdapter use a com.gensym.shell.util.DefaultWorkspaceFactoryImpl to generate instances of the WorkspaceDocument class whenever they create a workspace document. As described in "WorkspaceDocument" on page 210, a WorkspaceDocument uses the default menu bar and toolbar of the TW2MDIApplication.

If you want your application to provide context-sensitive menu bars and/or toolbars when a workspace document gains focus, you must create a custom `WorkspaceDocument` class and implement a `WorkspaceDocumentFactory` to generate instances of your custom workspace document. For details on how to do this, see:

- "Creating a Custom Workspace Document" on page 214.
- "Using Workspace Document Factories" on page 211.

Each class in your application that creates a workspace document needs to register your implementation of `WorkspaceDocumentFactory` to generate instances of your `WorkspaceDocument` type, rather than instances of a `WorkspaceDocument`.

Typically, you register the workspace document factory for a class in the application's constructor or `main` method to ensure the factory is set before the class creates any workspace documents.

To register the `WorkspaceDocumentFactory` with a class:

1. Create an instance of an implementation of `WorkspaceDocumentFactory`, which the class uses to generate `WorkspaceDocument` types.
2. In the class that generates workspace documents, call the method that sets its workspace document factory.

You may only call the method that sets the workspace document factory once for the class that requires it.

For example, the `WorkspaceCommands` class defines a method called `setWorkspaceDocumentFactory`, which sets the factory that the command uses for generating workspace documents.

Here is the method that the `Shell` class calls in its constructor to register the `com.gensym.shell.ShellWorkspaceDocumentFactoryImpl` for an instance of `WorkspaceCommands`:

```
private WorkspaceCommands wkspHandler;

private void registerWorkspaceDocumentFactory() {
    ShellWorkspaceDocumentFactoryImpl shellWkspDocFactory =
        new ShellWorkspaceDocumentFactoryImpl();
    if (wkspHandler != null)
        ((WorkspaceCommands)wkspHandler).
            setWorkspaceDocumentFactory(shellWkspDocFactory);
}
```

Similarly, the following code in the `Shell` class registers the workspace document factory for the `TW2MDIWorkspaceShowingAdapter`. The line of code that registers the factory appears in the `main` method.

```
private ConnectionManager connectionManager;
private TW2MDIWorkspaceShowingAdapter workspaceShowingListener = null;
private Shell application = new Shell(cmdLineArgs);

//Create adapter
workspaceShowingListener = new TW2MDIWorkspaceShowingAdapter
    (application.connectionManager);

//Register factory
workspaceShowingListener.setWorkspaceDocumentFactory
    (application.shellWkspDocFactory);
```

10

Using Shell Dialogs and UI Controls

Describes how to use the shell dialogs and UI controls, and provides a reference for each class.

Introduction

The `com.gensym.shell.dialogs` and `com.gensym.shell.util` packages provide two categories of classes, which you can use in your application:

- **Shell dialogs** — Standard dialogs for logging into G2.
- **Shell UI controls** — UI controls that display and let you switch the host and port of the current connection, and the current G2 user mode.

Each reference section in this chapter provides:

- A sample dialog or UI control.
- A general description of the dialog or UI control and any special behavior.

- The constructor or constructors, and the unique arguments to the public constructor.
- Example.

`LoginDialog` is a subclass of `StandardDialog`, which means it behaves like all standard dialogs, as described in Chapter 4, “Using Standard Dialogs” on page 71.

For a description of the common arguments to all standard dialog classes, see “Common Arguments to Standard Dialog Constructors” on page 76.

Packages Covered

com.gensym.shell.dialogs

```
LoginDialog
```

com.gensym.shell.util

```
HostPortPanel
UserModePanel
```

Relevant Demos

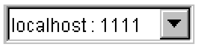
The following demos show examples of shell dialogs and UI controls:

- `singlecxnsdiapp`
- `singlecxnmdiapp`
- `multiplecxnmdiapp`

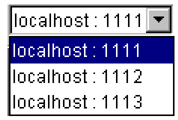
The demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/`

HostPortPanel



JLabel



JComboBox

Description

`HostPortPanel` provides a UI control that displays the currently open connection, which uses the host name and port number.

The constructor you use depends on whether the user is allowed to edit the user mode, and whether your application allows multiple connections to G2.

You embed this UI control in a toolbar or dialog to provide a user interface for displaying the current connection in a single or multiple connection application. You can also use this control in multiple connection applications to switch the G2 connection.

Constructor

`HostPortPanel` provides three constructors:

Use this constructor...	To create a dialog that...
`HostPortPanel()`	Displays the host and port of the current connection as static text in a `javax.swing.JLabel`. Use this constructor in single connection applications where the user is not allowed to switch connections.
`HostPortPanel (TwConnection connection)`	Displays the host and port of the current connection in a `javax.swing.JComboBox`. Use this constructor in single connection applications when the user is allowed to switch connections.
`HostPortPanel (ConnectionManager connectionMgr)`	Displays the host and port of the current connection, as well as a list of all open connections, in a `javax.swing.JComboBox`. Use this constructor in multiple connection applications when the user is allowed to switch connections.

Example

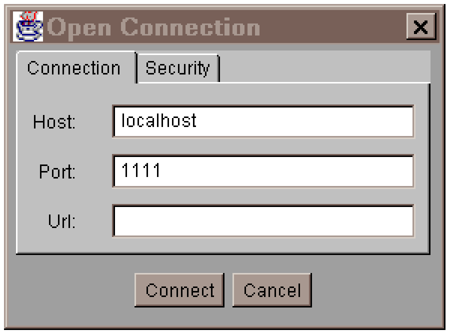
This example creates a `com.gensym.mdi.MDIToolBarPanel` that includes a `HostPortPanel` in a `com.gensym.ui.toolbar.ToolBar`. The example shows how to add the `HostPortPanel` to a multiple connection application, which allows the user to switch the current connection.

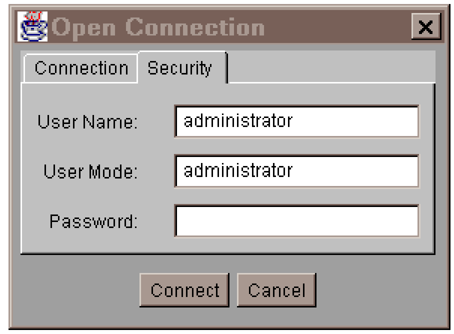
For more information on...	See...
Creating toolbar panels	"Creating an MDI Toolbar Panel" on page 197.
Adding buttons and panels to toolbars	"Adding All Command Keys" on page 124.

Here is the method that creates the toolbar panel, where the constructor for the `HostPortPanel` appears in bold:

```
private ConnectionManager connectionManager;

private MDIToolBarPanel createToolBarPanel() {

   //Create toolbar panel
   MDIToolBarPanel panel = new MDIToolBarPanel();

   //Create toolbar
   ToolBar tb = new ToolBar ();
   try {

      //Add HostPortPanel
      tb.add (new HostPortPanel(connectionManager));
      tb.add (javax.swing.Box.createGlue());
   } catch (G2AccessException e) {
      e.printStackTrace();
   }

   //Add toolbar to panel
   panel.add(tb);

   //Return panel
   return panel;
}
```

LoginDialog





Description

`LoginDialog` provides a tabbed dialog for connecting to G2 and logging on to a secure G2. It provides these two tab pages, both of which are editable, by default:

- Connection, for specifying the host, port, and URL.
- Security, for specifying the user name, user mode, and password.

To specify which tab pages are editable or read-only:

➔ Call this method:

```
setEditableTabPages(int tabPages, boolean isEditable)
```

Use one of the following variables to specify the `tabPages` argument:

```
CONNECTION_TAB_PAGE
SECURITY_TAB_PAGE
```

Use one of the following variables to specify the `isEditable` argument:

```
CONNECTION_AND_SECURITY_TAB_PAGES
NO_TAB_PAGES
```

A `boolean` value of `true` indicates that the specified tab pages are editable; a value of `false` indicates that the tab pages are read-only.

To specify which tab page is selected by default:

➔ `selectTabPage(int tabPage)`

You can call methods on a `LoginDialog` to get and set connection and login information, which the dialog uses to update information on the tab pages, as this table describes:

Call these methods...	To get and set these objects...	Which the dialog uses to update information on...
`getConnectionInformation` `setConnectionInformation`	`com.gensym.ntw.TwConnectionInfo`	The Connection tab page.
`getLoginRequest` `setLoginRequest`	`com.gensym.ntw.LoginRequest`	The Security tab page.

For information on these core classes, see these chapters in the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*:

- Chapter 5, "Using Connection Information Objects."
- Chapter 7, "Establishing a G2 Login Session."

Constructor

The `LoginDialog` constructor takes as its only arguments the common arguments to all standard dialogs.

For a description of the common arguments to all standard dialog classes, see "Common Arguments to Standard Dialog Constructors" on page 76.

Example

The `handleOpenConnectionCommand` method launches a `LoginDialog` to open a connection to a secure G2. The method sets the default value of the host, port, and URL by calling `setConnectionInformation`. It sets the default value of the user name, user mode, and password by calling `setLoginRequest`. The set methods take instance of a `com.gensym.ntw.TwConnectionInfo` and a `com.gensym.ntw.LoginRequest`, respectively, as arguments.

The dialog provides a localized text string as the dialog title. For details on using resource properties files, see Appendix A, "Localization" on page 331.

Here is the method that opens a connection by getting connection and login information from the `LoginDialog`:

```
TwConnectionInfo previousConnectionInfo =
    new TwConnectionInfo(brokerURL, hostName, portNumber);
LoginRequest previousLoginRequest =
    new LoginRequest(userMode_, userName_, password_);
private com.gensym.message.Resource i18nUI = Resource.getBundle
    ("com.gensym.demos.singlecxnmdiapp.UiLabels");
private com.gensym.shell.util.TW2Application application;
```

```
private void handleOpenConnection() {

    //Create LoginDialog
    if (loginDialog == null) {
      LoginDialog loginDialog = new LoginDialog
          (null, i18nUI.getString("OpenConnectionDialogTitle"),
           true, this);
    }

    //Set host, port, and URL
    loginDialog.setConnectionInformation(previousConnectionInfo);

    //Set user name, user mode, and password
    loginDialog.setLoginRequest(previousLoginRequest);

    //Specify that Security tab page is read-only
    loginDialog.setEditableTabPages(LoginDialog.SECURITY_TAB_PAGE,
                                    false);

    //Select Connection tab, by default
    loginDialog.selectTabPage(LoginDialog.CONNECTION_TAB_PAGE);

    //Launch dialog
    loginDialog.setVisible(true);
  }
```

The `openConnectionDialogDismissed` method implements the behavior of a `StandardDialogClient` that listens for the action event associated with closing the `LoginDialog`. The method takes a `LoginDialog` as its argument. It gets the value of the host and port text fields by calling `getConnectionInformation` on the dialog. It uses these values to open a connection, through a `com.gensym.ntw.TwGateway`.

For information on implementing a `StandardDialogClient`, see "Listening for Dialog Events" on page 77.

```
private void openConnectionDialogDismissed(LoginDialog dlg) {
   TwConnectionInfo newConnectionInfo =
      dlg.getConnectionInformation();

   try {
      String host = newConnectionInfo.getHost();
      String port = newConnectionInfo.getPort();
      TwAccess unloggedInConnection = TwGateway.openConnection(host,
                                                              port);
      previousConnectionInfo = newConnectionInfo;
      // The following call will fail if the G2 is secure.
      unloggedInConnection.login();
      TW2Application application =
         (TW2Application)GensymApplication.getApplication();
   }
   catch (G2AccessException e) {
      e.printStackTrace();
   }
   dlg.setVisible(false);
   }
```

The `dialogDismissed` method simply calls the `openConnectionDialogDismissed` method to implement the behavior of the standard dialog client.

```
public void dialogDismissed(StandardDialog dlg, int code) {
   if (dlg.wasCancelled()) return;
   openConnectionDialogDismissed((LoginDialog)dlg);
}
```

UserModePanel

Description

`UserModePanel` provides a `javax.swing.JComboBox` that displays the user mode of the current connection. Depending on how you construct the panel, the user can switch the user mode:

- First, by entering a new value in the text area,
- Then, by choosing the user mode from the list of available modes.

Note The UI control initializes with the current G2 user mode; it does not initialize with all available user modes.

The constructor you use depends on whether the user is allowed to edit the user mode and whether your application allows multiple connections to G2.

You embed this control in a toolbar or dialog to provide a user interface for displaying or switching the user mode of the current connection.

Constructor

`UserModePanel` provides three constructors:

Use this constructor...	To create a dialog that...
`UserModePanel()`	Displays the user mode of the current connection as static text. Use this constructor when the user is not allowed to edit the user mode.

Use this constructor...	To create a dialog that...
`UserModePanel (TwConnection connection, boolean allowUserModeAddition)`	Displays the user mode of the current connection and provides a list of previously entered user modes from which to choose. Use this constructor in single connection applications. Use the `boolean` argument to specify whether or not the user can enter a new value in the combo box.
`UserModePanel (ConnectionManager connectionMgr, boolean allowUserModeAddition)`	Displays the user mode of the current connection and provides a list of previously entered user modes from which to choose. Use this constructor in multiple connection applications. Use the `boolean` argument to specify whether or not the user can enter a new value in the combo box.

Example

This example creates a `com.gensym.mdi.MDIToolBarPanel` that includes a `UserModePanel` in a `com.gensym.ui.toolbar.ToolBar`. The example shows how to add the `UserModePanel` to a multiple connection application, which allows the user to switch the current user mode.

For more information on...	See...
Creating toolbar panels	“Creating an MDI Toolbar Panel” on page 197.
Adding buttons and panels to toolbars	“Adding All Command Keys” on page 124.

Here is the method that creates the toolbar panel, where the constructor for the `UserModePanel` appears in bold:

```
private ConnectionManager connectionManager;

private MDIToolBarPanel createToolBarPanel() {

    //Create a toolbar panel
    MDIToolBarPanel panel = new MDIToolBarPanel();

    //Create toolbar
    ToolBar tb = new ToolBar ();
    try {

        //Add UserModePanel with type-in capability
        tb.add (new UserModePanel(connectionManager, true));
    } catch (G2AccessException e) {
        e.printStackTrace();
    }

    //Add toolbar to panel
    panel.add(tb);

    //Return panel
    return panel;
}
```

11

Using Shell Commands

Describes commands that you use in an application shell to perform common tasks, such as connecting to G2, starting and pausing G2, getting named KB workspaces, and interacting with items on KB workspaces.

Introduction

The com.gensym.shell.commands package provides a number of built-in commands, called **shell commands**, which you can use directly in your application to perform standard interactions with G2. The Telewindows2 (TW2) Toolkit Java application shell uses all of these commands in its default and context-sensitive menu bars and toolbars.

This table describes and gives a page reference for each command:

Class	Description	See
ConnectionCommands	Connects to and disconnects from G2.	page 276
SwitchConnectionCommand	Switches the G2 connection in an application that supports multiple connections.	page 290
G2StateCommands and CondensedG2StateCommands	Changes the G2 run state.	page 283
EditCommands	Provides standard cut/copy/paste commands for editing items on a KB workspace.	page 279
ItemCommands	Performs standard G2 operations on items on a KB workspace, such as, lift to top, drop to bottom, enable, disable, and delete.	page 287
WorkspaceCommands	Creates a new KB workspace and gets a named KB workspace.	page 293
WorkspaceInstanceCommands	Performs operations on a KB workspace, such as editing its properties, selecting all items, and printing.	page 296
ZoomCommands	Scales a workspace view in or out, by a percentage, or to fit the workspace view.	page 299

Class	Description	See
`ExitCommands`	Exits the application.	page 281
`HelpCommands`	Displays a help dialog.	page 286
`TraceCommands`	Customizes how the application handles tracing and exceptions.	page 291

To use a shell command in your application, add the command to a command-aware container, as described in "Creating Command-Aware Containers" on page 122.

Each reference section in this chapter provides:

- A general description of the command and any special behavior.
- The available command keys, their behavior, and their iconic representation.
- The constructors for each command.
- The command availability in applications that support single connections and multiple connections, where relevant.

Command Keys

Each command defines a final static variable for each command key, for example, `TW_CONNECT`.

You refer to this key when you add individual command keys to a command-aware container.

You also use the command key as a key into a resource properties file to localize command text.

Constructors

You use different versions of the constructor, depending on the type of application and whether your application supports single or multiple connections to G2.

Many commands provide two constructors, which take one or more of the following types of arguments:

- A frame:
  - `java.awt.Frame`
  - `com.gensym.mdi.MDIFrame`

- A single document interface (SDI) or multiple document interface (MDI) application:
  - `com.gensym.shell.util.TW2Application`
  - `com.gensym.shell.util.TW2MDIApplication`
- A connection or connection manager:
  - An implementation of `com.gensym.ntw.TwAccess`, such as `TwGateway`
  - `com.gensym.shell.util.ConnectionManager`

For example, `ConnectionCommands` provides two versions of its constructor for use with either SDI or MDI applications, respectively:

- `ConnectionCommands(TW2Application app)`
- `ConnectionCommands(TW2MDIApplication app)`

Similarly, `ExitCommands` provides two versions of its constructor for use with any application frame, and a single connection or multiple connection application, respectively:

- `ExitCommands(Frame frame, TwAccess connection)`
- `ExitCommands(Frame frame, ConnectionManager connectionManager)`

For information on using...	See...
`java.awt.Frame`	"UiApplication" on page 229.
`TW2Application`	"Creating Single Document Interface Applications" on page 247.
`TW2MDIApplication`	"Creating Multiple Document Interface Applications" on page 251.
`TwGateway`	"Will the Application Support Single or Multiple Connections to G2?" on page 236.
`ConnectionManager`	"Creating and Managing Connections to G2" on page 236.

Availability

Some commands are always available, whereas others become available or unavailable when certain events occur, such as when the connection context of a multiple connection application changes. The description of each command defines when it becomes available and unavailable.

For more information about command availability, see “Delivering Command Events By Setting Properties” on page 135.

Packages Covered

com.gensym.shell.commands

```
CondensedG2StateCommands
ConnectionCommands
CreationCommands
EditCommands
ExitCommands
G2StateCommands
HelpCommands
ItemCommands
SwitchConnectionCommand
TraceCommands
WorkspaceCommands
WorkspaceInstanceCommands
ZoomCommands
```

Note `G2StateCommand` and `CondensedG2StateCommand` are documented together under `G2StateCommand`.

Relevant Demos

The following demos show examples of shell commands:

- `singlecxnmdiapp`
- `multiplecxnmdiapp`
- `multiplecxnsdiapp`

The demos are located in this directory, depending on your platform:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\demos\`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/demos/`

ConnectionCommands

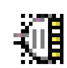
ConnectionCommands provides command keys for connecting to and disconnecting from G2.

You can use ConnectionCommands in both single and multiple connection applications.

Note Multiple connection applications must use a com.gensym.shell.util. ConnectionManager to maintain open connections to G2.

Command Keys

ConnectionCommands provides two command keys and icons:

This command key...	Performs this action...	And defines this icon...
TW_CONNECT	Displays a dialog for specifying the host, port, and URL of a G2 to which to connect, as well as the user name, user mode, and password to log on to a secure G2. Click OK in the dialog to connect to G2, using the specified connection and login information.	
TW_DISCONNECT	Displays a dialog with a list of open G2 connections. Choose a connection and click OK to close the selected connection.	

Constructors

You can use `ConnectionCommands` in both SDI and MDI applications by using the appropriate version of its constructor:

If you are creating a...	Use this version of the constructor...
Single document interface application	`ConnectionCommands(TW2Application app)`
Multiple document interface application	`ConnectionCommands(TW2MDIApplication app)`

Availability

The command keys have this availability in single connection applications:

This command key...	Is available when...	Is unavailable when...
`TW_CONNECT`	The current connection is `null`.	The current connection is not `null`.
`TW_DISCONNECT`	The current connection is not `null`.	The current connection is `null`.

The command keys have this availability in multiple connection applications:

This command key...	Is available when...	Is unavailable when...
`TW_CONNECT`	Always.	Never.
`TW_DISCONNECT`	At least one connection exists.	No connection exists.

CreationCommands

`CreationCommands` provides command keys for creating items on a KB workspace.

You can use `ConnectionCommands` in both single and multiple connection applications.

Command Keys

`CreationCommands` provides two command keys and no icons:

This command key...	Performs this action...
NEW_ITEM	Displays a palette of items, which you can drag and drop onto a workspace view. To edit the items on the palette, right-click the palette and choose Edit Classes, then add system-defined and/or user-defined classes to the palette.
NEW_BEAN	Creates a G2 bean on a KB workspace. **Note:** This feature is currently not supported.

Constructors

You can use `CreationCommands` in any type of user-interface application by using this constructor:

```
CreationCommands()
```

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
NEW_ITEM	A workspace view is selected.	A workspace view is not currently selected.
NEW_BEAN	A workspace view is selected.	A workspace view is not currently selected.

EditCommands

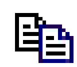
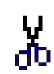
EditCommands provides command keys for standard cut/copy/paste actions for interacting with G2 items on KB workspaces.

Note You can paste the cut or copied item into workspace views from the same connection only; you cannot paste the item into a workspace view from a different connection or another TW2 Toolkit application.

The commands apply to the selected items in the current workspace view. To change the current workspace view programmatically, call the command's setWorkspaceView method.

Command Keys

EditCommands provides three command keys:

This command key...	Performs this action...	And defines this icon...
COPY_SELECTION	Copies the currently selected item to the clipboard.	
CUT_SELECTION	Places the currently selected items in the clipboard buffer.	
PASTE_SELECTION	Transfers the clipboard buffer to the current workspace view.	

Constructors

You can use `EditCommands` in SDI or MDI applications. If you are creating an MDI application, you typically add this command to a context-sensitive menu bar associated with a subclass of `com.gensym.shell.util.WorkspaceDocument`. The command provides two constructors:

If you are creating a...	Use this version of constructor...
Single document interface application	`EditCommands()`
Multiple document interface application	`EditCommands(MDIFrame parentFrame)`

For information on creating context-sensitive menu bars and workspace documents, see Chapter 8, "Using Telewindows2 Toolkit MDI Documents" on page 207.

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
CUT_SELECTION	An item is selected.	No item is selected.
COPY_SELECTION	An item is selected.	No item is selected.
PASTE_SELECTION	An item is on the clipboard for the current connection.	No item is on the clipboard.

ExitCommands

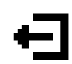
`ExitCommands` provides a single command key for exiting the application. The command closes all open connections before exiting.

You can use this command in both single and multiple connection applications. You maintain the current connection differently depending on the type of connection:

In applications that support...	Do this...
Single connections	You must update the command with the current connection by calling the `setConnection` method on `ExitCommands`.
Multiple connections	The `com.gensym.shell.util.ConnectionManager` takes care of maintaining the current connection for you.

Command Keys

`ExitCommands` provides a single command key and icon:

This command key...	Performs this action...	And defines this icon...
`EXIT`	Closes any open G2 connections and exits the application.	

Constructors

You can use `ExitCommands` in both single and multiple connection applications, by using the appropriate version of the constructor:

If you are creating a...	Use this version of the constructor...
Single connection application	`ExitCommands(Frame frame, TwAccess connection)`
Multiple connection application	`ExitCommands(Frame frame, ConnectionManager connectionManager)`

Note In a single connection application, if the connection is not known at the time at which the command is created, you can pass `null` as the argument to the constructor. To update the command with the connection information when it is available, call `setConnection` on the `ExitCommands` instance.

Availability

The command key has this availability:

This command key...	Is available...	Is unavailable...
`EXIT`	Always.	Never.

G2StateCommands and CondensedG2StateCommands

`G2StateCommands` provides command keys for starting, pausing, resuming, resetting, and restarting G2 from the client. You can choose between two versions of this command:

Use this command...	If you want to provide...
`G2StateCommands`	Separate command keys for starting, pausing, resuming, restarting, and resetting G2.
`CondensedG2StateCommands`	A single command key for starting, pausing, and resuming, which switches to the appropriate command key, depending on the context.

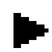
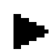
All G2 run state changes affect the current connection.

You can use this command in both single and multiple connection applications. You maintain the current connection differently depending on the type of connection:

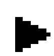
In applications that support...	Do this...
Single connections	You are responsible for updating the command with the current connection by calling the `setConnection` method on `G2StateCommands`.
Multiple connections	The `com.gensym.shell.util.ConnectionManager` takes care of maintaining the current connection for you.

Command Keys

`G2StateCommands` provides five command keys and icons:

This command key...	Performs this action...	And defines this icon...
`PAUSE`	Pauses G2 when running.	
`RESET`	Resets G2.	
`RESTART`	Restarts G2.	
`RESUME`	Resumes G2 when paused.	
`START`	Starts G2 when reset.	

`CondensedG2StateCommands` provides three command keys:

This command key...	Performs this action...	And defines these icons...
`START_PAUSE_OR_ RESUME`	Starts G2 when reset, pauses G2 when running, or resumes G2 when paused.	 
`RESET`	Resets G2.	
`RESTART`	Restarts G2.	

Constructors

You can use `G2StateCommands` and `CondensedG2StateCommands` in both single and multiple connection applications by using the appropriate version of the constructor:

If you are creating a...	Use this version of each constructor...
Single connection application	`G2StateCommands(TwAccess connection)` `CondensedG2StateCommands(TwAccess connection)`
Multiple connection application	`G2StateCommands (ConnectionManager connectionManager)` `CondensedG2StateCommands (ConnectionManager connectionManager)`

Note In a single connection application, if the connection is not known at the time at which the command is created, you can pass `null` as the argument to the constructor. To update the command with the connection information when it is available, call `setConnection` on the `G2StateCommands` or `CondensedG2StateCommands` instance.

Availability

The command's availability reflects the current state of the G2 server that corresponds to the current connection.

HelpCommands

`HelpCommands` provides a single command key that displays information about the TW2 Toolkit default application shell in a dialog. The command launches an instance of a `com.gensym.dlg.AboutDialog`.

Note This command launches an About dialog with text specific to the TW2 Toolkit default application shell, which you cannot edit. To create an About dialog for your application, create and launch an instance of an `AboutDialog`, as described in `AboutDialog` on page 95.

Command Keys

`HelpCommands` provides a single command key and no icon:

This command key...	Performs this action...
`ABOUT`	Displays the About dialog for the TW2 Toolkit default application shell, which contains help text within a scrollable text region.

Constructors

You can use `HelpCommands` in any type of user-interface application by calling its constructor:

```
HelpCommands(Frame frame)
```

The constructor takes any subclass of `java.awt.Frame` as its argument, including any of the TW2 application foundation classes provided in the `com.gensym.shell.util` package.

Availability

The command key has this availability:

This command key...	Is available...	Is unavailable...
`ABOUT`	Always.	Never.

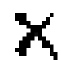
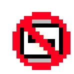
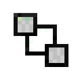
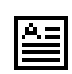
ItemCommands

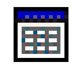
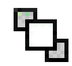
`ItemCommands` provides numerous command keys that correspond to the G2 system-defined user menu choices for items on a KB workspace.

The commands apply to the items in the current workspace view. To change the current workspace view programmatically, call the `setWorkspaceView` method on the command.

Command Keys

`ItemCommands` provides the following command keys:

Command Key	Action	Icon
DELETE_SELECTION	Permanently deletes the selected item(s) in the current workspace view.	
DISABLE_SELECTION	Disables the selected item(s).	
DROP_SELECTION_TO_BOTTOM	Drops the selected item(s) to the bottom of the drawing order.	
EDIT_ITEM_TEXT	Launches the native text editor for editing the text attribute of the selected item(s).	
ENABLE_SELECTION	Enables the selected item(s).	

Command Key	Action	Icon
ITEM_PROPERTIES	Displays the item properties dialog for the selected item(s).	
LIFT_SELECTION_TO_TOP	Lifts the selected item(s) to the top of the drawing order.	

Constructors

You can use `ItemCommands` in SDI or MDI applications. If you are creating an MDI application, you typically add this command to a context-sensitive menu bar associated with a subclass of `com.gensym.shell.util.WorkspaceDocument`. The command provides two constructors:

If you are creating a...	Use this version of constructor...
Single document interface application	`ItemCommands()`
Multiple document interface application	`ItemCommands(MDIFrame parentFrame)`

For information on creating context-sensitive menu bars and workspace documents, see Chapter 8, “Using Telewindows2 Toolkit MDI Documents” on page 207.

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
DELETE_SELECTION	At least one item is selected.	No item is selected.
ENABLE_SELECTION	All selected items are disabled.	A selected item(s) is(are) enabled.
DISABLE_SELECTION	All selected items are enabled.	A selected item(s) is(are) disabled.
DROP_SELECTION_TO_BOTTOM	At least one item is selected.	No item is selected.
LIFT_SELECTION_TO_TOP	A single item is selected.	No item is selected.
EDIT_ITEM_TEXT	The selected item has a text attribute.	The selected item has no text attribute.
ITEM_PROPERTIES	A single item is selected.	No item is selected.

SwitchConnectionCommand

SwitchConnectionCommand provides a command key that switches between open G2 connections in a multiple connection application.

As new connections are opened, the command keeps track of the open connections and presents them to the user in a cascading submenu.

This command is only applicable in multiple connection applications. The application is responsible for maintaining open G2 connections through a com.gensym.shell.util.ConnectionManager.

Command Keys

SwitchConnectionCommand provides a single command key and no icon:

This command key...	Performs this action...
TW_SWITCH_CONNECTION	Displays a cascading submenu with a list of all open G2 connections. The submenu updates dynamically when a new connection is opened or an existing connection is closed. To switch the current connection, select a connection from the submenu.

Constructors

Because you can only switch connections in a multiple connection application, SwitchConnectionCommand provides a single constructor, which takes a ConnectionManager as its argument:

```
SwitchConnectionCommand(ConnectionManager connectionMgr)
```

Availability

The command key has this availability:

This command key...	Is available when...	Is unavailable when...
TW_SWITCH_CONNECTION	A connection exists.	No connection exists.

TraceCommands

`TraceCommands` provides a subcommand that launches a dialog for customizing the trace level and configuring exception handling. The command also provides command keys for enabling and disabling different levels of tracing.

Command Keys

`TraceCommands` provides four command keys and no icons:

This command key...	Performs this action...
`CUSTOMIZE`	Displays a dialog for setting the trace keys, trace level, and trace messages.
`EXCEPTIONS`	Enables or disables application-level exception printing.
`GLOBAL`	Enables or disables all tracing.
`TRACE`	Displays a cascading submenu that includes the `EXCEPTIONS`, `GLOBAL`, and `CUSTOMIZE` command keys.

For information about tracing and debugging, see the *G2 JavaLink User's Guide*.

Constructors

You can use `TraceCommands` in any type of user-interface application by using this constructor:

```
TraceCommands(Frame frame)
```

Availability

The command keys have this availability:

This command key...	Is available...	Is unavailable...
`CUSTOMIZE`	Always.	Never.

This command key...	Is available...	Is unavailable...
EXCEPTIONS	Always.	Never.
GLOBAL	Always.	Never.
TRACE	Always.	Never.

WorkspaceCommands

`WorkspaceCommands` provides command keys for getting a named KB workspace and creating a new KB workspace.

The command adds a workspace view to the appropriate container, depending on the type of application, as this table describes:

If you are creating a...	The command does this...
Single document interface application	Adds the workspace view to the center of the current application frame.
Multiple document interface application	Uses a `com.gensym.shell.util.DefaultWorkspaceDocumentFactoryImpl` to create a `com.gensym.shell.util.WorkspaceDocument`, and adds the workspace view to the workspace document.

To add the workspace view to a subclass of `WorkspaceDocument`:

- Call the `setWorkspaceDocumentFactory` method on the command.
- Provide an implementation of `com.gensym.shell.util.WorkspaceDocumentFactory` as the argument to the `set` method.

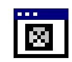
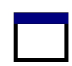
For information on creating workspace document types and using workspace document factories, see Chapter 8, “Using Telewindows2 Toolkit MDI Documents” on page 207.

You can use this command in both single and multiple connection applications. You maintain the current connection differently depending on the type of connection:

In applications that support...	Do this...
Single connections	You are responsible for updating the command with the current connection by calling the `setConnection` method on `WorkspaceCommands`.
Multiple connections	The `com.gensym.shell.util.ConnectionManager` takes care of maintaining the current connection for you.

Command Keys

`WorkspaceCommands` provides two command keys and icons:

This command key...	Performs this action...	And defines this icon...
`GET_WORKSPACE`	Displays a `com.gensym.dlg.SelectionDialog` with a scrolling list of all named KB workspaces. Select a KB workspace and click OK to download the selected workspace and display it in the appropriate container, depending on the type of application.	
`NEW_WORKSPACE`	Creates an unnamed KB workspace and adds it to a workspace document, according to the registered workspace document factory.	

Constructors

You can use `WorkspaceCommands` in both single and multiple connection application, by using the appropriate version of the constructor:

If you are creating a...	Use this version of the constructor...
Single connection application	`WorkspaceCommands(Frame frame, TwAccess connection)`
Multiple connection application	`WorkspaceCommands(Frame frame, ConnectionManager connectionMgr)`

Note In a single connection application, if the connection is not known at the time at which the command is created, you can pass `null` as the argument to the constructor. To update the command with the connection information when it is available, call `setConnection` on the `WorkspaceCommands` instance.

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
GET_WORKSPACE	The current connection is not `null`.	The current connection is `null`.
NEW_WORKSPACE	The current connection is not `null`.	The current connection is `null`.

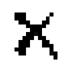
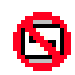
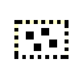
WorkspaceInstanceCommands

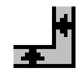
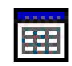
`WorkspaceInstanceCommands` provides command keys for the G2 system-defined menu choices for KB workspaces, such as printing, shrink wrapping, deleting, and displaying the properties dialog.

All the actions of the command apply to the current workspace view. To change the current workspace view programmatically, call the `setWorkspaceView` method on the command.

Command Keys

`WorkspaceInstanceCommands` provides the following command keys and icons:

This command key...	Performs this action...	And defines this icon...
DELETE_WORKSPACE	Deletes the selected KB workspace.	
DISABLE_WORKSPACE	Disables the selected KB workspace.	
ENABLE_WORKSPACE	Enables the selected KB workspace.	
PRINT_WORKSPACE	Displays a standard dialog for specifying the print destination, page range, copies, and properties for printing the selected KB workspace. Click OK to send the selected KB workspace to the specified destination.	
SELECT_ALL_WORKSPACE_ITEMS	Selects all items on the selected KB workspace.	

This command key...	Performs this action...	And defines this icon...
SHRINK_WRAP_WORKSPACE	Shrink wraps the selected KB workspace.	
WORKSPACE_PROPERTIES	Displays the properties dialog for the selected KB workspace.	

Constructors

You can use `WorkspaceInstanceCommands` in SDI or MDI applications. If you are creating an MDI application, you typically add this command to a context-sensitive menu bar associated with a subclass of `com.gensym.shell.util.WorkspaceDocument`. The command provides two constructors:

If you are creating a...	Use this version of constructor...
Single document interface application	`WorkspaceInstanceCommands()`
Multiple document interface application	`WorkspaceInstanceCommands (MDIFrame parentFrame)`

For information on creating context-sensitive menu bars and workspace documents, see Chapter 8, “Using Telewindows2 Toolkit MDI Documents” on page 207.

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
PRINT_WORKSPACE	A workspace document has focus.	No workspace document has focus.
DELETE_WORKSPACE	A workspace document has focus.	No workspace document has focus.
DISABLE_WORKSPACE	The workspace is enabled.	The workspace is disabled.

This command key...	Is available when...	Is unavailable when...
ENABLE_WORKSPACE	The workspace is disabled.	The workspace is enabled.
SELECT_ALL_WORKSPACE_ITEMS	A workspace document has focus.	No workspace document has focus.
SHRINK_WRAP_WORKSPACE	A workspace document has focus.	No workspace document has focus.
WORKSPACE_PROPERTIES	A workspace document has focus.	No workspace document has focus.

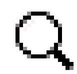
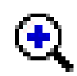
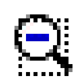
ZoomCommands

ZoomCommands provides command keys for setting the zoom scale of the current workspace view, zooming in, and zooming out.

All the actions of the command apply to the current workspace view. To change the current workspace view programmatically, call the setWorkspaceView method on the command.

Command Keys

ZoomCommands provides the following command keys and icons:

This command key...	Performs this action...	And defines this icon...
ZOOM	Launches a Zoom dialog for choosing from one of a number of standard zoom scales or entering a specific percentage to zoom.	
ZOOM_IN	Scales the workspace to 1.2 times its current size, by default.	
ZOOM_OUT	Scales the workspace to .8 times its current size, by default.	

Constructors

You can use ZoomCommands in any UI application, with or without default values:

If you wish to use...	Use this version of constructor...
A default zoomInAmount of 1.2 and a default zoomOutAmount of 0.8.	`ZoomCommands(Frame frame)`
Your own zoom amounts	`ZoomCommands(Frame frame, double[] values, String[] labels, boolean includeZoomToFit, boolean includeZoomPercent, double zoomInAmount, double zoomOutAmount)`

Availability

The command keys have this availability:

This command key...	Is available when...	Is unavailable when...
ZOOM	A workspace document has focus.	No workspace document has focus.
ZOOM_IN	A workspace document has focus.	No workspace document has focus.
ZOOM_OUT	A workspace document has focus.	No workspace document has focus.

12

Understanding the Telewindows2 Toolkit Shell

Describes the implementation of the Telewindows2 Toolkit default application shell for Java, which is an example of a multiple connection MDI application.

Introduction

Telewindows2 (TW2) Toolkit includes a default application shell that you can use as an example of the kind of G2 client application you can build in Java. This shell is referred to as the **TW2 Toolkit shell**, or just the **shell**.

The TW2 Toolkit shell exists to illustrate how a UI developer might create a client user interface for interacting with G2. The TW2 Toolkit shell is an example of a multiple connection, multiple document interface (MDI) application; however, the techniques it uses are applicable for building any type of G2 client application.

In addition, you can use the shell as a simple user interface for connecting to multiple G2 servers, viewing KB workspaces, editing the attributes of items through item properties dialogs, and controlling the G2 run state.

For a walk-through of the TW2 Toolkit shell user interface, see Chapter 2, “Guided Tour of the Telewindows2 Toolkit Shell” on page 33.

Telewindows2 Toolkit Default Application Shell Features

The Telewindows2 Toolkit shell provides these features, which the referenced sections describe in detail:

This feature...	Is described in detail in...
A **Telewindows2 Toolkit MDI application** capable of displaying and manipulating multiple workspaces views	“Creating Multiple Document Interface Applications” on page 251.
A **context changed listener**, which updates the status bar when the context changes	“Listening for Changes in the Current Connection Context” on page 242.
A **default menu bar** that supports built-in commands for connecting to multiple G2 servers, changing the G2 run mode, and getting KB workspaces	• “Creating Command-Aware Containers” on page 122. • Chapter 11, “Using Shell Commands” on page 271.

This feature...	Is described in detail in...
A **default toolbar panel** with several commonly used commands and UI controls	• "Creating Command-Aware Containers" on page 122. • Chapter 10, "Using Shell Dialogs and UI Controls" on page 259. • Chapter 11, "Using Shell Commands" on page 271.
Multiple connections to G2 through a `ConnectionManager`	"Creating and Managing Connections to G2" on page 236.
A **custom workspace document** that provides a context-specific menu bar, and an associated workspace document factory that generates the custom workspace document	Chapter 8, "Using Telewindows2 Toolkit MDI Documents" on page 207.
Localized text	Appendix A, "Localization."

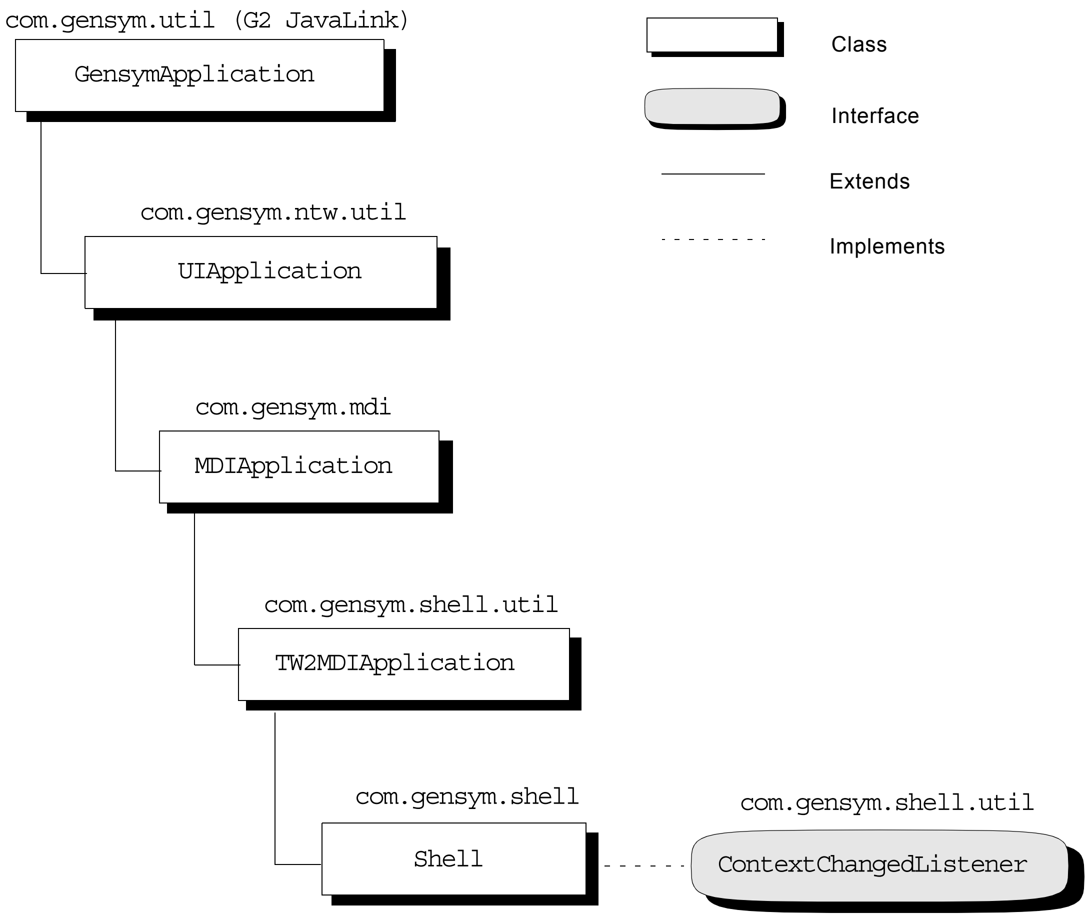
The Shell Class

The TW2 Toolkit shell:

- **Extends `com.gensym.shell.util.TW2MDIApplication`**, which means it is a multiple document interface application.
- **Implements `com.gensym.shell.util.ContextChangedListener`**, which means it listens for changes in the connection context, as maintained by the `com.gensym.shell.util.ConnectionManager`.

Inheritance Structure

The following figure shows the inheritance structure of the `Shell` class and the package location of each of its superior classes.



For information on these application foundation classes, see “Application Foundation Classes” on page 227.

Source Code

This section shows the complete `Shell` source code with comments.

The source code is located in this file in your TW2 Toolkit product directory:

NT:	`%SEQUOIA_HOME%\classes\com\gensym\shell\Shell.java`
UNIX:	`$SEQUOIA_HOME/classes/com/gensym/shell/Shell.java`

See the sections that follow for explanations of each major feature.

```
package com.gensym.shell;

import java.awt.Image;
import java.awt.Toolkit;
import com.gensym.jgi.G2AccessException;
import com.gensym.message.Resource;
import com.gensym.message.Trace;
import com.gensym.mdi.MDIFrame;
import com.gensym.mdi.MDIManager;
import com.gensym.mdi.MDIDocument;
import com.gensym.mdi.MDIStatusBar;
import com.gensym.mdi.MDIToolBarPanel;
import com.gensym.ntw.LoginRequest;
import com.gensym.ntw.TwAccess;
import com.gensym.ntw.TwConnectionAdapter;
import com.gensym.ntw.TwConnectionEvent;
import com.gensym.ntw.TwConnectionInfo;
import com.gensym.shell.commands.*;
import com.gensym.shell.util.*;
import com.gensym.ui.RepresentationConstraints;
import com.gensym.ui.menu.CMenu;
import com.gensym.ui.menu.CMenuBar;
import com.gensym.ui.toolbar.ToolBar;
import com.gensym.util.Symbol;
import javax.swing.SwingConstants;
import javax.swing.UIManager;

public class Shell extends TW2MDIApplication
    implements ContextChangedListener {

    //******************
    // Private Variables
    //******************

    //G2 Menu System variable
    private static final Symbol GMS_ = Symbol.intern ("GMS");

    //Resource variable to support localization
    private static Resource i18nUI =
        Resource.getBundle("com.gensym.shell.Messages");

    //Application frame variable
    private MDIFrame frame = null;

    //Variable for creating and managing multiple connections
    private ConnectionManager connectionManager;
```

```
//UI container variables
private MDIStatusBar statusBar;
private MDIToolBarPanel toolBarPanel;
private CMenuBar menuBar;
private RepresentationConstraints menuConstraints =
   new RepresentationConstraints
       (RepresentationConstraints.TEXT_AND_ICON,
        SwingConstants.LEFT,
        SwingConstants.CENTER,
        SwingConstants.RIGHT,
        SwingConstants.CENTER);

//User mode variable
private Symbol userMode;

//Exception listener and handler for login failures
//from command line arguments
private ShellModeListener modeListener = new ShellModeListener();
private TW2LoginFailureHandler loginFailureHandler;

//Command variables that the shell adds to its UI containers
ConnectionCommands connectionHandler;
SwitchConnectionCommand switchConnectionHandler;
CondensedG2StateCommands g2StateHandler;
ExitCommands exitHandler;
HelpCommands helpHandler;
TraceCommands traceHandler;
WorkspaceCommands wkspHandler;

//Variable responsible for creating the correct subclass of
//WorkspaceDocument in which to display workspace views
private WorkspaceDocumentFactory shellWkspDocFactory;

//***********************************
// Constructor and Constructor Method
//***********************************

public Shell (String[] cmdLineArgs) {

   //Parse the command line arguments
   super(cmdLineArgs);

   //Create the Shell container
   createShell();

private void createShell() {

   //Create the MDIFrame
   frame = createFrame(i18nUI.getString("ShellTitle"));

   //Set the current frame
   setCurrentFrame(frame);

   //Create the ConnectionManager
   connectionManager = new ConnectionManager();
```

```
    //Add ConnectionManager as listener for ContextChangedEvents
    connectionManager.addContextChangedListener(this);

    //Create the UI components
    cseateUiComponents();

    //Register workspace document factory
    registerWorkspaceDocumentFactory();

    //Set up login failure handler
    loginFailureHandler = new ShellLoginFailureHandler();
}

//*************************
// TW2MDIApplication Methods
//*************************

//getConnectionManager returns ConnectionManager
//in multiple connection applications
public ConnectionManager getConnectionManager(){
    return connectionManager;
}

//getConnection returns null in multiple connection applications
public TwAccess getConnection(){
    return null;
}

//setConnection has no implementation in multiple
//connection applications
public void setConnection(TwAccess connection){}

//***********************************
// Application Frame and UI Components
//***********************************

//Create the UI components
protected void createUiComponents() {
    frame.setStatusBar(statusBar = createStatusBar());
    frame.setDefaultMenuBar(menuBar = createMenuBar());
    frame.setDefaultToolBarPanel(toolBarPanel =
                                   createToolBarPanel());
}
```

```
//Create the MDI frame
private MDIFrame createFrame(String title) {

    //Create the MDI frame
    MDIFrame frame =  new MDIFrame(title);

    //Set the logo to be the Gensym logo
    Image image = Toolkit.getDefaultToolkit().
        getImage(this.getClass().getResource("gensym_logo.gif"));
    if (image != null)
        frame.setIconImage(image);

    return frame;
}

//Create the menu bar
protected CMenuBar createMenuBar() {
    CMenuBar mb = new CMenuBar();

    // Create the FILE menu
    mb.add(createFileMenu());

    // Create the G2 menu
    mb.add(createG2Menu());

    // Create the HELP menu
    mb.add(createHelpMenu());

    return mb;
 }

//Create the toolbar panel
protected MDIToolBarPanel createToolBarPanel() {

    //Create the toolbar panel
    MDIToolBarPanel panel = new MDIToolBarPanel();

    //Create the first toolbar, which contains buttons
    ToolBar tb = new ToolBar();

    //Add buttons and separators to the first toolbar
    tb.add(workspaceHandler, WorkspaceCommands.GET_KBWORKSPACE);
    tb.addSeparator();
    tb.add(connectionHandler);
    tb.addSeparator();
    tb.add(g2StateHandler);

    //Add the first toolbar to the toolbar panel
    panel.add(tb);

    //Create a second toolbar
    ToolBar tb2 = new ToolBar();
```

```
    //Add a HostPortPanel for switching the connection
    try {
        tb2.add(new HostPortPanel(connectionManager));
        tb2.add(javax.swing.Box.createGlue());
    } catch(G2AccessException ex) {
        Trace.exception (ex);
    }

    //Add a UserModePanel for switching the user mode
    try {
        tb2.add(new UserModePanel(connectionManager, true));
    } catch(G2AccessException ex) {
    Trace.exception (ex);
    }

    //Add the second toolbar to the toolbar panel
    panel.add(tb2);

    //Return the toolbar panel
    return panel;
}

//Create the status bar
protected MDIStatusBar createStatusBar() {
    MDIStatusBar sb = new MDIStatusBar();
    return sb;
}

//*******************
// Menus and Toolbars
//*******************

//Create File menu
private CMenu createFileMenu() {

    //Create instance of a pulldown menu and get the
    //menu bar text from the short resource bundle
    CMenu menu = new CMenu (i18nUI.getString("ShellFileMenu"));

    //Create instances of commands and add to
    //menu with constraints, add separators
    wkspHandler = new WorkspaceCommands(frame,
                                    connectionManager);
    menu.add(wkspHandler, WorkspaceCommands.GET_KBWORKSPACE,
             menuConstraints);
    exitHandler = new ExitCommands(frame, connectionManager);
    menu.addSeparator();
    menu.add(exitHandler, menuConstraints);

    //Return menu
    return menu;
}
```

```
//Create G2 menu
protected CMenu createG2Menu() {
    CMenu menu = new CMenu(i18nUI.getString("ShellG2Menu"));
    switchConnectionHandler = new
        SwitchConnectionCommand(connectionManager);
    menu.add(switchConnectionHandler);
    connectionHandler = new ConnectionCommands(this);
    menu.add(connectionHandler, menuConstraints);
    g2StateHandler = new
        CondensedG2StateCommands(connectionManager);
    menu.addSeparator();
    menu.add(g2StateHandler, menuConstraints);
    return menu;
}

//Create Help menu
private CMenu createHelpMenu() {
    CMenu menu = new
        CMenu(i18nUI.getString("ShellHelpMenu"));
    helpHandler = new HelpCommands(frame);
    menu.add(helpHandler);
    traceHandler = new TraceCommands(frame);
    menu.addSeparator();
    menu.add(traceHandler);
    return menu;
}

//*************************
// WorkspaceDocumentFactory
//*************************

//Register WorkspaceDocumentFactory with workspace handler
private void registerWorkspaceDocumentFactory() {
    shellWkspDocFactory =
        new ShellWorkspaceDocumentFactoryImpl();
    if (wkspHandler != null)
        ((WorkspaceCommands)wkspHandler).
            setWorkspaceDocumentFactory(shellWkspDocFactory);
}

//******************************
// ContextChangedListener Method
//******************************

public void currentConnectionChanged(ContextChangedEvent e) {

    //Get current connection from ContextChangedEvent
    TwAccess connection = e.getConnection();

    //Set status bar status to null if no connection
    if (connection == null)
        setStatusBarStatus (i18nUI.getString("ShellNoConnection"),
                                               null);
```

```
    //Set status bar status to current connection and
    //user mode if connection exist
    else {
        if (e.isConnectionNew())
            connection.addTwConnectionListener (new modeListener());
        if (connection.isLoggedIn()) {
            try {
                userMode = connection.getUserMode();
                setStatusBarStatus(connection.toShortString(),
                                   userMode);
            } catch (G2AccessException ex) {
            Trace.exception(ex);
            }

        //Set status bar to null if not logged in
        } else {
            setStatusBarStatus(connection.toShortString(), null);
        }
    }
}

//******************
// Status Bar Method
//******************

private void setStatusBarStatus(String connection, Symbol mode) {
    String status = connection;
    if (mode != null)
        status = status + "  " + mode.toString().toLowerCase();
    statusBar.setStatus(status);
}

class ShellModeListener extends TwConnectionAdapter {
    public void loggedIn (TwConnectionEvent e) {
        userMode = e.getUserMode ();
        TwAccess connection = (TwAccess) e.getSource ();
        setStatusBarStatus(connection.toShortString(), userMode);
    }

    public void userModeChanged (TwConnectionEvent e) {
        userMode = e.getUserMode ();
        TwAccess connection = (TwAccess) e.getSource ();
        TwAccess currentCxn = connectionManager.getCurrentConnection();
        if (connection != null && currentCxn != null &&
            connection == currentCxn) {
            setStatusBarStatus(connection.toShortString(), userMode);
        }
    }
}
```

```
//************
// Main Method
//************

public static void main(String[] cmdLineArgs) {

//Set the look and feel of Swing classes
try {
   UIManager.setLookAndFeel
       (UIManager.getSystemLookAndFeelClassName());
}
catch (Exception ex) {
   throw new com.gensym.util.UnexpectedException(ex);
}

//Create an instance of the class
Shell application = new Shell(cmdLineArgs);

//Handle command line arguments for UiApplication
String title = getTitleInformation();
if (title != null)
   application.frame.setTitle(title);

String geometry = getGeometryInformation();
if (geometry != null)
   setBoundsForFrame(application.frame, geometry);

//Open a connection, using command line arguments
TwAccess unloggedInConnection = null;
TW2MDIWorkspaceShowingAdapter workspaceShowingListener = null;

try {

   //Get ConnectionManager from application
   ConnectionManager connectionMgr =
      application.getConnectionManager();

   //Get TwConnectionInfo from application
   TwConnectionInfo connectionInfo = getG2ConnectionInformation();

   //Create a WorkspaceShowingListener to respond to programmatic
   //show and hide actions in G2
   workspaceShowingListener = new TW2MDIWorkspaceShowingAdapter
       (application.connectionManager);

   if (connectionInfo != null) {
      //Set connection information in ConnectionCommands
      application.connectionHandler.
         setPreviousConnectionInformation(connectionInfo);
```

```
            //Open a connection and make a LoginRequest
            connectionMgr.openConnection(connectionInfo);
            LoginRequest loginRequest = getLoginRequest();
            if (loginRequest != null){
               //Set login information in the ConnectionCommands
               application.connectionHandler.
                  setPreviousLoginRequest(loginRequest);
               unloggedInConnection =
                  connectionMgr.getCurrentConnection();
               if (unloggedInConnection != null)
                  unloggedInConnection.login(loginRequest);
            }
         }
         //Register the WorkspaceDocumentFactory for the
         //WorkspaceShowingListener
         workspaceShowingListener.
            setWorkspaceDocumentFactory(application.shellWkspDocFactory);
      }
      //Handle exceptions
      catch (G2AccessException gae) {
         Trace.exception (gae);
         application.loginFailureHandler.
            handleLoginFailureException(gae, unloggedInConnection);
      }
      //Make the frame visible
      application.frame.setVisible(true);
   }
}
```

Constructor and Constructor Method

The `Shell` constructor and the `createShell` method perform these tasks:

- **Parses the command-line arguments.**

  The constructor calls the constructor for its superior class, which parses command-line arguments, and calls the `createShell` method, which performs the actual tasks of the constructor.

  By calling `super(cmdLineArgs)`, the constructor lets:

  – `TW2MDIApplication` parse these command-line arguments:

    `-url, -host, -port`

  – `UIApplication` parse these command-line arguments:

    `-title, -geometry`

  – `GensymApplication` parse and handle these command line arguments:

    `-language, -country, -variant`

  For the details of how the `Shell` class handles command-line arguments, see “Main Method” on page 322.

- **Creates the `MDIFrame`.**

  The constructor is responsible for creating the `MDIFrame` and getting its title from the message resource bundle.

  For details, see “Application Frame and UI Components” on page 316.

- **Sets the current frame.**

  The constructor sets the current frame of the `UiApplication` to make it available to other features of the application via the `getCurrentFrame` method.

  For more information, see “Creating and Setting the Frame in an MDI Application” on page 252.

- **Creates the `ConnectionManager`.**

  Because the TW2 Toolkit shell supports multiple G2 connections, the constructor creates a `ConnectionManager` for creating and managing open connections.

  For an explanation of how the TW2 Toolkit shell creates and manages connections, see “Main Method” on page 322.

- **Adds the `ConnectionManager` as a `ContextChangedListener`.**

  By adding the `ConnectionManager` as a listener for `ContextChangedEvents`, the manager gets notified when the connection context changes. The application updates the status bar when the context changes.

  For an explanation of the `ContextChangedListener` method, see “ContextChangedListener Method” on page 321.

  For an explanation of the method that updates the status bar when the context changes, see “Create the Status Bar” on page 317.

- **Creates the UI components.**

  The constructor is responsible for creating the user interface components that the application uses. The TW2 Toolkit shell creates a menu bar, toolbar, and status bar.

  For details, see “Application Frame and UI Components” on page 316.

- **Registers the `WorkspaceDocumentFactory`** with **`WorkspaceCommands`**.

  The constructor is responsible for creating and setting the `WorkspaceDocumentFactory`, which determines the type of workspace document to create when displaying workspace views in a workspace document.

  For details, see “Register WorkspaceDocumentFactory” on page 321.

- **Sets up login failure handler.**

  Creates an instance of a `ShellLoginFailureHandler`, which encapsulates login error handling for the TW2 Toolkit shell.

  For details, see the source code for this class:

  ```
  com.gensym.shell.ShellLoginFailureHandler
  ```

TW2MDIApplication Methods

Because the TW2 Toolkit shell extends `TW2MDIApplication`, it must implement these abstract methods for getting the `ConnectionManager` and the connection:

- **`getConnectionManager`** — Returns the `com.gensym.shell.util.ConnectionManager`, because `Shell` is a multiple connection application.
- **`getConnection`** — Returns `null`, because `Shell` is a multiple connection application.
- **`setConnection`** — Has no implementation because the TW2 Toolkit shell manages connections through a `ConnectionManager`.

For more information, see “Implementing Abstract Methods to Manage Connections” on page 244.

Application Frame and UI Components

The constructor through the private `createShell` method calls these two private methods:

- **`createFrame`** — Creates an instance of a `com.gensym.mdi.MDIFrame` by calling the `createFrame` method, passing a localized text string as the title.
- **`createUiComponents`** — Creates these UI components:
  - Menu bar
  - Toolbar
  - Status bar

To create default UI components as part of the application frame, the TW2 Toolkit shell calls `set` methods on the `MDIFrame` for each type of UI component. The methods create a default menu bar, a default toolbar, and a status bar. The shell displays the default menu bar and toolbar when no document has focus. The shell updates the status bar when the connection context changes.

Each `set` method takes an instance of the associated UI component as its argument. The `set` methods create these UI components by calling `create` methods for each type of container, which the application defines.

The following sections describe the implementation of each `create` method.

Create the Menu Bar

The `MDIFrame` has a single default menu bar, which is an instance of a `com.gensym.ui.menu.CMenuBar`.

The `createMenuBar` method:

- Creates an instance of a `CMenuBar`.
- Calls the `add` method on the menu bar to add each pulldown menu.
- Returns the menu bar.

The `add` method takes an instance of a `CMenu` as its argument, which is the pulldown menu associated with the top-level menu choice on the menu bar.

Each `add` method dynamically creates the pulldown menus by calling a `create` method for each pulldown menu, which the application defines.

For a description of one these `create` methods, see “Create File, G2, and Help Menu” on page 318.

Create the Toolbar Panel

The MDIFrame is associated with a single default toolbar panel, which is an instance of a com.gensym.mdi.MDIToolBarPanel.

The toolbar panel, in turn, contains two toolbars, which are instances of a com.gensym.ui.toolbar.ToolBar.

The TW2 Toolkit shell's toolbar panel consists of these two toolbars:

- A toolbar of icons that let you connection to and disconnect from G2, change the G2 run state, and get a named KB workspace.
- A toolbar of choice boxes that let you switch between multiple connections and switch the G2 user mode.

The createToolBarPanel method performs these tasks:

- Adds the two toolbars to the toolbar panel by calling the add method on the MDIToolBarPanel.
- Calls an add method for each set of toolbar buttons, which is a method that the application defines to create an instance of the command that contains the toolbar buttons.
- Calls the add method directly on the ToolBar to add the two choice boxes.
- Adds separators to a toolbar by calling the addSeparator method on the ToolBar.
- Returns the toolbar panel.

For a description of these add methods, see "Create Toolbar" on page 319.

Create the Status Bar

The MDIFrame has a status bar, which is an instance of a com.gensym.mdi.MDIStatusBar.

The application updates the current connection and user mode of the status bar when the current connection context changes by being a ContextChangedListener.

For information on the implementation of the ContextChangedListener method that updates the status bar, see "ContextChangedListener Method" on page 321.

Menus and Toolbars

The TW2 Toolkit shell supports two command-aware containers to which it adds commands:

- A menu bar, which consist of three default menus:
  - File
  - G2
  - Help
- A toolbar panel

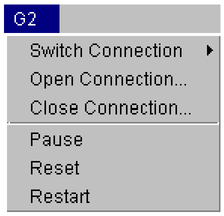
For information on creating menus and toolbars, see "Creating Command-Aware Containers" on page 122.

Create File, G2, and Help Menu

The method that creates the menu bar, `createMenuBar`, calls three `create` methods, which create instances of each pulldown menu and consist of commands located in the `com.gensym.shell.commands` package:

- **`createFileMenu`** — Consists of these commands:
  - `WorkspaceCommands`, which defines the Get Workspace and New Workspace commands.
  - `ExitCommands`, which defines the Exit command.
- **`createG2Menu`** — Consists of these commands:
  - `SwitchConnectionCommand`, which defines the Switch Connection command.
  - `ConnectionCommands`, which defines the Open Connection and Close Connection commands.
  - `CondensedG2StateCommands`, which defines the Start/Pause/Resume, Reset, and Restart commands.
- **`createHelpMenu`** — Consists of these commands:
  - `HelpCommands`, which defines the About command.
  - `TraceCommands`, which defines the Customize, Exceptions, Global, and Trace commands.

Each command consists of one or more command keys. For example, the G2 pulldown menu looks like this:



For information on these commands, see Chapter 11, “Using Shell Commands” on page 271.

Each pulldown menu’s `create` method performs these tasks:

- Creates an instance of a `com.gensym.ui.menu.CMenu`, passing a localized text string as the menu label.
- Creates an instance of each command as a handler.
- Adds the handler to the pulldown menu by calling the menu’s `add` method.
- Adds separators to the pulldown menu, using its `addSeparator` method.
- Returns the pulldown menu.

Create Toolbar

The method that creates the toolbar panel, `createToolBarPanel`, creates two toolbars:

- The first toolbar consists of toolbar buttons.
- The second toolbar consists of UI controls for viewing and changing the current connection and G2 user mode.

The first toolbar calls various private `add` methods to create toolbar buttons from shell commands.

The second toolbar calls the `add` method directly on the toolbar to add the UI controls.

To see the method that adds the toolbar to the panel, see “Create the Toolbar Panel” on page 317.

First Toolbar

The first toolbar calls the following methods to create instances of commands in the `com.gensym.shell.commands` package, where each command consists of several command keys:

- **`addWorkspaceCommandsToolBarButtons`** — Creates an instance of `WorkspaceCommands`.
- **`addConnectionCommandsToolBarButtons`** — Creates an instance of `ConnectionCommands`.
- **`addG2StateCommandsToolBarButtons`** — Creates an instance of `CondensedG2StateCommands`.

The methods each take an instance of a `com.gensym.ui.toolbar.ToolBar` as their argument, which the method that creates the toolbar panel provides.

Each toolbar button's `add` method performs these tasks:

- Creates an instance of each command as a handler.
- Adds the handler to the toolbar by calling the toolbar's `add` method.

For example, the `addConnectionCommandsToolBarButtons` method adds toolbar buttons for the command keys specified in `ConnectionCommands`. `ConnectionCommands` takes as its argument an instance of a `com.gensym.shell.util.ConnectionManager`, which is created elsewhere in the application.

The resulting toolbar looks like this:

For information on these commands, see Chapter 11, "Using Shell Commands" on page 271.

Second Toolbar

The second toolbar calls the add method directly on the `Toolbar` to add instances of these UI controls, both in the `com.gensym.shell.util` package:

- `UserModePanel` — Lets the user switch the G2 user mode of the current connection.
- `HostPortPanel` — Lets the user switch the current connection.

The resulting toolbar looks like this:

For information on these UI controls, see Chapter 10, "Using Shell Dialogs and UI Controls" on page 259.

Register WorkspaceDocumentFactory

The constructor for the `Shell` class is responsible for registering the `com.gensym.shell.util.WorkspaceDocumentFactory` with the instance of `com.gensym.shell.commands.WorkspaceCommands` to specify the type of workspace document the command creates when the user chooses Get Workspace or New Workspace.

To do this, the constructor calls the private `registerWorkspaceDocumentFactory` method, which performs these tasks:

- Creates an instance of a `ShellWorkspaceDocumentFactoryImpl`, which the application uses to create workspace documents.
- Sets the `ShellWorkspaceDocumentFactoryImpl` for the `WorkspaceCommand` handler.

For information on setting the `WorkspaceDocumentFactory` for the `TW2MDIWorkspaceShowingAdapter`, see “Main Method” on page 322.

For an explanation of the workspace document factory and associated workspace document, see “ShellWorkspaceDocument and ShellWorkspaceDocumentFactory” on page 325.

ContextChangedListener Method

Because the `Shell` class supports multiple connection through a `com.gensym.shell.util.ConnectionManager`, it implements the `com.gensym.shell.util.ContextChangedListener` interface. The `Shell` constructor adds itself as a listener so it receives notification of changes in the current connection context as maintained by the `ConnectionManager`.

For details, see “Constructor and Constructor Method” on page 314.

By implementing this interface, the `Shell` receives notification from the `ConnectionManager` whenever the current connection closes or whenever a new connection becomes the current connection. When the current connection context changes, the `ConnectionManager` invokes the listener’s `currentConnectionChanged` method, which performs these tasks:

- Checks to see if the TW2 Toolkit client is still connected to a G2 session.
- If no connection exists, calls the `setStatusBarStatus` method, passing in:
  - A key for the connection, which the application looks up in the short resource properties file at run time to determine the string to display.
  - `null` for the user mode.
- If a connection exists, adds the connection as a listener for `TwConnectionEvents` and calls the `setStatusBarStatus` method, passing the current connection and user mode as the arguments.

For a description of the setStatusBarStatus method, see “Create the Status Bar” on page 317.

For information on ContextChangedListener, see “Listening for Changes in the Current Connection Context” on page 242.

Status Bar Method

The setStatusBarStatus private method updates the status bar with the current connection and user mode. It does this by creating an inner class, TwModeListener, which extends com.gensym.ntw.TwConnectionAdapter, an adapter class for com.gensym.ntw.TwConnectionListener.

TwModeListener implements these two methods:

- **loggedIn** — Sets the status of the status bar to the current status when the user logs on to a G2 connection.
- **userModeChanged** — Gets the current user mode from the TwConnectionEvent when the user changes the user mode.

The methods convert the user mode and connection to a string, and displays them in the status bar by calling setStatus.

Main Method

The body of the main method for the Shell class performs these tasks:

- **Sets the look and feel of Swing classes.**

  Because the TW2 Toolkit shell is built on top of Java Swing components, you can choose to set the look and feel of these swing classes to reflect the look-and-feel of your operating system.

- **Creates an instance of the Shell class.**

  The main method is responsible for creating an instance of the class.

- **Handles command line arguments for UiApplication.**

  The main method is responsible for handling the -title and -geometry command-line arguments, which the com.gensym.core.UiApplication parses from the command line.

  For more information, see “UiApplication” on page 229.

- **Handles command-line arguments for `TW2MDIApplication`** to **makes a connection through a `ConnectionManager`**.

  The TW2 Toolkit shell supports creating multiple connections to a secure G2 by handling the `-host`, `-port`, and `-brokerUrl` command-line arguments, and the `-userName`, `-userMode`, and `-password` command-line arguments. The shell uses these command-line arguments to make a secure G2 connection by performing these tasks in it `main` method:

  – Creates a `ConnectionManager` by calling `getConnectionManager` on `com.gensym.shell.util.TW2MDIApplication`.

  – Gets the connection information from the command line by calling `getG2ConnectionInformation` on `TW2MDIApplication`.

  – Sets the default connection information for the `com.gensym.shell.dialogs.LoginDialog`, which `com.gensym.shell.commands.ConnectionCommands` launches when the user connects to G2.

  – Creates a connection by calling `openConnection` on the `com.gensym.shell.util.ConnectionManager`, passing in the `com.gensym.ntw.TwConnectionInfo` as argument.

  – Creates a `com.gensym.ntw.LoginRequest` by calling `getLoginRequest` on `TW2MDIApplication`.

  – If the `LoginRequest` succeeds, sets the default login information in the `LoginDialog`, which `ConnectionCommands` launches when the user creates a connection.

  – Makes a login request by calling `login` on `ConnectionManager`, passing in a `LoginRequest`.

  For more information, see “TW2MDIApplication” on page 232.

- **Creates a `WorkspaceShowingListener`**.

  The `main` method creates an instance of a `com.gensym.shell.util.TW2MDIWorkspaceShowingAdapter` so the application is notified when G2 programmatically shows or hides KB workspaces.

  For more information, see “Listening for Programmatic Show and Hide KB Workspace Events in an MDI Application” on page 254.

- **Registers the default `WorkspaceDocumentFactory`** for programmatic KB workspace showing events in G2.

  The application is responsible for determining the type of workspace document the `TW2MDIWorkspaceShowingAdapter` uses when it creates a workspace document based on programmatic show KB workspace events in G2. To do this, it uses a `ShellWorkspaceDocumentFactoryImpl`, which creates a `ShellWorkspaceDocument`. Calling the `setWorkspaceDocumentFactory` method on the adapter sets the workspace document factory so the adapter generates instances of ShellWorkspaceDocument, instead of WorkspaceDocument.

The TW2 Toolkit shell constructor creates an instance of the workspace document factory when it sets the workspace document factory for WorkspaceCommands.

For the method that creates the workspace document factory, see "Register WorkspaceDocumentFactory" on page 321.

For more information about the workspace document and associated factory, see "ShellWorkspaceDocument and ShellWorkspaceDocumentFactory" on page 325.

- **Handles exceptions.**

  The ConnectionManager methods that creates a connection and logs on to a secure G2 throw these exceptions:

  – openConnection throws ConnectionTimedOutException, G2AccessInitiationException, and G2AccessException, all in the com.gensym.jgi package, which is part of G2 JavaLink.

  – login throws G2AccessException.

  If an exception occur, the application creates an error dialog, providing localized title and message text.

  For information on formatting and localizing message text, see Appendix A, "Localization."

- **Makes the frame visible.**

  Finally, the main method makes the application frame visible.

  For more information, see "Creating and Setting the Frame in an MDI Application" on page 252.

ShellWorkspaceDocument and ShellWorkspaceDocumentFactory

The default application shell provides its own `MDIDocument` type and associated `WorkspaceDocumentFactory`, as follows:

Class	Description
`ShellWorkspaceDocument`	A subclass of `com.gensym.shell.util.WorkspaceDocument` that provides a context-specific menu bar for interacting with workspace views.
`ShellWorkspaceDocumentFactoryImpl`	An implementation of the `com.gensym.shell.util.WorkspaceDocumentFactory` interface that generates instances of a `ShellWorkspaceDocument`.

For more information, see:

- “Using MDI Document Types” on page 209.
- “Using Workspace Document Factories” on page 211.

ShellWorkspaceDocument

`ShellWorkspaceDocument` provides these features:

- A constructor that creates a workspace document, given a connection and KB workspace.
- A context-specific menu bar that is the same as the default menu bar for the `MDIFrame` of the `Shell` class, with the addition of these menus:
  - Edit menu
  - Item menu
  - Workspace menu
  - Window menu
- A toolbar panel that is the default toolbar panel of the `MDIFrame`.
- A title that appends the connection information to the name passed in as an argument when the workspace document is created.

The following sections show the relevant sections of the source code. To see the complete source code, see the source code for this class:

```
com.gensym.shell.ShellWorkspaceDocument
```

Constructor

Here is the constructor:

```
public ShellWorkspaceDocument(TwAccess connection, KbWorkspace wksp){
    super(connection, wksp, menuBar, windowMenu,
            frame.getDefaultToolBarPanel());
}
```

Menu Bar

This method generates the menu bar, where each `create` method that gets added returns an instance of a `CMenu`:

```
private static CMenuBar menuBar = createMenuBar();

private static CMenuBar createMenuBar(){
    menuBar = new CMenuBar();
    CMenu fileMenu = createFileMenu();
    menuBar.add(fileMenu);
    menuBar.add(createEditMenu());
    menuBar.add(createItemMenu());
    menuBar.add(createWorkspaceMenu());
    menuBar.add(createG2Menu());
    menuBar.add(createWindowMenu());
    menuBar.add(createHelpMenu());
    return menuBar;
}
```

Window Menu

This method generates the Window menu, providing a localized text string for the menu label:

```
private static CMenu createWindowMenu(){
    CMenu windowMenu = new CMenu(i18nUI.getString("ShellWindowMenu"));
    windowMenu.add(frame.getManager().getTilingCommand());
    return windowMenu;
}
```

Title

This method overrides the `setTitle` method in the superior class:

```
public void setTitle(String name){
    super.setTitle(name+" ("+getConnection().toShortString()+")");
}
```

ShellWorkspaceDocumentFactory

Here is the complete source code for the `ShellWorkspaceDocumentFactoryImpl` that generates a `ShellWorkspaceDocumentFactory`:

```
package com.gensym.shell;

import com.gensym.shell.util.WorkspaceDocumentFactory;
import com.gensym.shell.util.WorkspaceDocument;
import com.gensym.ntw.TwAccess;
import com.gensym.classes.KbWorkspace;

public class ShellWorkspaceDocumentFactoryImpl
    implements WorkspaceDocumentFactory{
    public WorkspaceDocument createWorkspaceDocument
        (TwAccess connection, KbWorkspace workspace){
            return new ShellWorkspaceDocument(connection, workspace);
    }
}
```

Part IV

Appendices

Localization

The examples in this guide use standard Java techniques for localizing applications. G2 JavaLink, the foundation upon which Telewindows2 Toolkit is built, supports localization by providing these classes:

- `com.gensym.core.GensymApplication` — Parses and handles these command line arguments that support localization:

  ```
  -language
  -country
  -variant
  ```

- `com.gensym.message.Resource` — A helper class that extends `java.util.ResourceBundle`, which supports formatting of message text and debugging of resource bundles.

This guide shows examples of localizing the following application text:

- Textual descriptions of commands.
- Application titles.
- Dialog text and titles.
- Error and message text.

Resources associated with commands typically use variables named `shortResource` and `longResource`, while resources associated with other text typically use variables whose name includes `i18n`. You initialize a resource by calling `getBundle` on the `Resource`, providing a fully qualified class name as a string, for example:

```
Resource i18nUI = Resource.getBundle("com.gensym.shell.Messages")
```

To localize a text string, you call `getString` on a `Resource`.

For information on...	See...
`GensymApplication`	`GensymApplication`
`com.gensym.message.Resource`	G2 JavaLink API

Deploying Your Application

Telewindows2 Toolkit only requires the Java Development Kit (JDK) to run in a development environment; it does not require the JDK to run in a deployment environment. Deploying a TW2 Toolkit application requires only the Java Runtime Environment (JRE), which you can redistribute.

Additionally, deployment does not require the `.com.gensym.properties` file.

At runtime, TW2 Toolkit can sometimes need to generate new Java classes if a class it needs is not on the client's disk. If you want TW2 Toolkit to save the classes created to disk, set the `com.gensym.class.user.repository` system property before connecting to G2. One way to do this is in the Java command line. The syntax is:

```
java -D<propertyName>=<DirPath> <class> <arguments>
```

where:

`<propertyName>` is `com.gensym.class.user.repository`

`<DirPath>` is the root directory location to which JavaLink should export user-defined classes.

For example, on NT:

```
java -Dcom.gensym.class.user.repository=c:\Program Files\gensym\
 g2-6.1\javalink\classes com.gensym.shell.Shell
 -host localhost -port 1111
```

Note The location specified for the `com.gensym.class.user.repository` must be included in your `CLASSPATH`.

Required Library Files

TW2 Toolkit deployment requires these JAR files:

JAR file	Product
coreui.jar sequoia.jar	From Telewindows2 Toolkit
classtools.jar core.jar javalink.jar	From G2 JavaLink

TW2 Toolkit deployment requires these dynamic link libraries:

Platform	DLL
Intel	JgiInterface.dll
Sparcsol	libJgiInterface.so

The JAR files must be on the client machine and specified in the CLASSPATH of the VM, by adding them to the CLASSPATH environment variable.

If an application is running in 3-tier mode:

- The middle tier requires the JAR files and the dynamic link libraries.
- The third tier client requires the JAR files.

To ensure that the VM is able to load the dynamic link libraries when running in either default (2-tier) mode or the middle tier of 3-tier mode, do one of the following:

- Set this system property to point to the location of the libraries:

  ```
  java.library.path
  ```

- Specify the location of the libraries in the appropriate environment variable:

Platform	Environment Variable
Intel	PATH
Sparcsol	LD_LIBRARY_PATH

Required Files for Beans Created with BeanXporter

Using the BeanXporter, you can convert Microsoft ActiveX components into Java Beans. Beans created with BeanXporter require these files:

- `ax2jbeans.jar`
- `JavaContainer.dll`

Glossary and Index

Glossary

A

abstract command: A default implementation of the `com.gensym.ui.Command` interface, which supports these features:

- Notifies listeners of command events, such as changes in the state or availability of the command.
- Supports get and set methods for command properties.
- Supports localization of textual descriptions.

See also command and structured command.

application foundation class: A class upon which you can build any type of G2 client application, using Telewindows2 Toolkit. To create a G2 client application, you extend the application foundation class that meets your requirements and implement its abstract methods. *See also* UI application, single document interface (SDI) application, multiple document interface (MDI) application, graphical UI classes, and shell class.

C

client: An application that runs on any platform and interacts through a network connection to view and manipulate data in a server. Telewindows2 Toolkit applications are client applications for connecting to a G2 server. *See* view data, manipulate data, and server.

command code: An integer that determines which dialog button the user has clicked. When implementing a `StandardDialogClient`, you might need to test the command code to determine the behavior of a standard dialog when it is dismissed. *See* standard dialog.

command: An action that the end user can perform through a user interface. A command is separate from the user interface that represents it. You represent commands in command-aware containers such as menus and toolbars. Commands notify listeners when one of its properties, such as state or availability, is set. *See* command-aware container. *See also* abstract command.

command key: A string that represents a single action of a command. A command may perform one or more actions by providing multiple command keys in its constructor. You specify command keys in a command information object, which you provide in the command's constructor. *See* abstract command.

command-aware container: A UI container that knows how to add a command, using a version of the add method. Command-aware containers represent commands appropriately depending on the type of container and whether the add method specifies representation constraints. Telewindows2 Toolkit supports menus and toolbars as command-aware containers. Command-aware containers are listeners for command events. *See* command.

custom dialog: An application-specific dialog you create by:

- Extending one of the standard dialog classes to customize the buttons, icon, or behavior of the dialog.
- Extending the superior class of all standard dialogs to customize the elements that appear in the dialog.
- Using Telewindows2 Toolkit dialog components in a Java programming environment to override automatically generated item properties dialogs and create any other type of dialog.

See standard dialog and properties dialog.

G

G2 JavaLink: The underlying technology that enables Telewindows2 Toolkit components to access and manipulate data in a G2 server, and to represent G2 items as components in a native, client application. For information on G2 JavaLink, see the *G2 JavaLink User's Guide*.

graphical user interface (UI) class: A class that provides a UI container or support for standard UI actions. Telewindows2 Toolkit provides a variety of graphical UI classes, including: commands and structured commands, menus and toolbars, standard informational and input dialogs, and multiple document interface frames and child documents. *See also* application foundation class and shell class.

I

informational dialog: A standard dialog that provides information to the user, such as errors, warnings, messages, questions, or help. Most informational dialogs have an icon and a single button for dismissing the dialog. *See* standard dialog. *See also* input dialog.

input dialog: A standard dialog that accepts input from the user, such as a dialog with text fields or a dialog with a list of items from which the user can select one or more items. Input dialogs provide OK and Cancel buttons, by default. *See* standard dialog. *See also* informational dialog.

J

Java Abstract Windowing Toolkit (AWT) and Java Foundation Classes (JFC): Java packages that provide the superior classes upon which the Telewindows2 Toolkit graphical user interface classes are built. *See* graphical UI classes.

Java application developer: A Java developer who builds applications in a pure Java programming environment. *See also* Java AWT and JFC, and Java programming.

Java programming: When developing Telewindows2 Toolkit applications, you need to be familiar with these aspects of Java programming:

- Properties, events, and methods of classes and interfaces.
- The Java 1.1. event model.
- Internationalization.
- Java AWT and Java Swing classes.

See Java application developer, and Java AWT and JFC.

For information on Java programming, see `www.java.sun.com` or any Java reference.

M

manipulate data: To modify data in the G2 server through a native, client user interface. *See also* view data.

modal dialog: A dialog that the user must dismiss before performing any other action in the application. *See also* standard dialog.

multiple connection application: An application that allows multiple connections to different G2 servers. You use a `ConnectionManager` to create and manage multiple connections. *See also* single connection application.

multiple document interface (MDI) application: An application that contains multiple child frames, or documents, for displaying and manipulating G2 data. Telewindows2 Toolkit provides an application foundation class that you can extend for creating MDI applications that manage connections to G2. *See also* SDI application.

N

native: Conforms to the “look-and-feel” of the window system on which the UI application is running. Telewindows2 Toolkit applications are native applications. *See* client and server.

P

properties dialog: The dialog associated with an item in a workspace view, which corresponds to the G2 attribute table. You display the properties dialog of an item from its popup menu. By default, workspace views generate item properties dialogs automatically; however, you can create custom item properties dialogs by using Telewindows2 Toolkit components. *See* workspace view and custom dialog.

S

separator: A horizontal bar in a menu and a vertical gap in a toolbar, which you can add explicitly or by creating a structured command. *See* command-aware container and structured command.

server: A running G2 executable, which is the source of all data that users view and manipulate through a native, client user interface. In a Telewindows2 Toolkit application, G2 is the server. *See* view data, manipulate data, and client.

shell: *See* TW2 Toolkit default application shell.

shell class: A class that defines the Telewindows2 Toolkit default application shell. The source code for shell classes is available for you to use as an example of the kind of application you can build. *See* TW2 Toolkit default application shell.

shell command: A class that supports common interactions with G2, such as creating a connection, changing the G2 run state, and creating and getting a KB workspace.

shell dialog: A standard dialog that supports common interactions with G2 from a client application, such as logging on to G2, and configuring message and error tracing. *See also* shell UI control, standard dialog, and custom dialog.

shell UI control: A UI control that provides support for common UI interactions with a G2 server, such as displaying and switching the host, port, and user mode of the current connection. *See also* shell dialog.

single connection application: An application that connects to a single G2 server. You use a `com.gensym.ntw.TwGateway` to create single connections. *See also* multiple connection application.

single document interface (SDI) application: An application that contains a single frame in which to display and manipulate G2 data, typically through a workspace view. *See also* MDI application.

standard dialog: A dialog class that you create to provide informational dialogs and dialogs that accept user input. Standard dialogs provide standard buttons and icons appropriate to the particular type of dialog, which you can customize. *See* informational dialog and input dialog. *See also* custom dialog.

structured command: A set of related actions with a hierarchical structure and/or a particular grouping, such as might appear in a menu with a cascading submenu.

The contents of a structured command can update dynamically. *See also* command.

syntax-guided text editor: The text editor that appears when you edit an attribute of an item in a workspace view that requires G2 syntax. You launch the text editor from an item properties dialog. *See* properties dialog.

T

Telewindows2 (TW2) Toolkit component: A Java Beans component that provide the basic support for connecting to a G2 server, displaying and manipulating data through a workspace view, handling the associated events, and representing G2 attribute values. For information on Telewindows2 Toolkit components, see the *Telewindows2 Toolkit Java Developer's Guide: Components and Core Classes*.

Telewindows2 (TW2) Toolkit default application shell: A Telewindows2 Toolkit application for making multiple connections to G2, and displaying and manipulating G2 data through a workspace view. You can use this shell as an example of the kind of G2 client application you can build by using Telewindows2 Toolkit, and as a simple user interface for running G2 applications from a client. *See also* shell class.

U

user interface (UI) application: A visual application for interacting with a G2 server through a client. If you are creating a UI application, the application is responsible for creating and maintaining the application frame, as well as creating and managing G2 connections.

user interface (UI) development: The general technique of constructing a user interface by adding Java components to containers and arranging those components by using layout managers. Java programmers who are building G2 client applications should be familiar with UI design techniques. *See* UI.

user interface (UI): Any kind of visual application that allows users, which includes end users or developers, to interact with data through user interface containers such as commands, menus, toolbars, and dialogs. Telewindows2 Toolkit allows you to create client UI applications for viewing and manipulating data in a G2 server. *See also* graphical UI classes and UI development.

V

view data: To display a visual representation of any type of G2 data, such as a workspace view, item properties dialog, or custom dialog. *See also* manipulate data.

W

workspace document: A type of Telewindows2 Toolkit document that contains a workspace view for use in MDI applications. A workspace document provides its own context-sensitive menu bar, which the application automatically swaps in when the document gains focus.

workspace document factory: A factory that generates any type of workspace document. *See* workspace document.

workspace view: A Telewindows2 Toolkit component that provides a client representation of a KB workspace. Telewindows2 Toolkit applications typically display workspace views within an application frame or child document of an MDI application. *See* workspace document.

Index

D

E

O

P

Q

R

S

Z