Sametime

Version 9.0

Sametime 9.0
Software Development Kit
Java Toolkit Tutorial

Edition Notice

Note: Before using this information and the product it supports, read the information in "Notices."

This edition applies to version 9.0 of IBM Sametime (program number 5725–M36) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Introduction

Intended Audience

This tutorial is intended for Java™ developers who want to learn how to use the IBM® Lotus® Sametime® Java Toolkit. It provides a quick and easy way for developers to learn about Sametime and how to Sametime-enable new or existing Java applications and applets with the Sametime Java Toolkit. Developers experienced with the previous version of the toolkit should first refer to the Readme in order to see the differences between this and the previous version. Developers experienced with earlier versions of the toolkit should use this tutorial to understand the changes and new features in the toolkit.

This tutorial does not include information about general Java programming. For more information about the Java language and Java programming, go to http://java.sun.com/

Requirements

For information about software requirements for the Sametime Java Toolkit, see releaseNotes.txt, included with the toolkit.

Although applications developed with this toolkit will work when run on Sametime 2.x or later servers, toolkit services that require features new to this release will not function. In particular, the code examples should be run on the latest version of the Sametime server.

How to Use this Tutorial

Part I – Introduction

The chapters in Part I introduce Sametime and its features and provide an overview of the Sametime Java Toolkit.

> **Chapter 2** describes the major Sametime services from an end-user perspective. This chapter is intended for readers who are less familiar with Sametime and would like a quick overview of Sametime capabilities. If you are already familiar with Sametime, skip this chapter and start from Chapter 3.
>
> **Chapter 3** provides an overview of the Sametime Java Toolkit architecture and services. The chapter explains how to run the sample code in this tutorial and includes a "Hello World" sample that demonstrates how to log in to the Sametime server with the toolkit.

Part II – Community Services Tutorial

Part II introduces the Sametime Java Toolkit and provides sample code to build a simple Sametime Buddy List applet through a series of steps. These steps cover the basics of using the Sametime Java Toolkit and include many of the Community Services.

> **Chapter 4** demonstrates how to log in to Sametime and listen to login notifications.
>
> **Chapter 5** demonstrates how to create a live list of Sametime users using the Awareness List AWT component.

Chapter 6 demonstrates how to change your online status and to initiate and join all types of Sametime meetings.

Chapter 7 demonstrates how to build a simple Buddy List application that stores the contact list on the Sametime server.

Part III – Places Tutorial

Part III describes advanced features of the Sametime Java Toolkit, including the Places Services. Sample code shows how to use the Places Services to build a meeting applet.

Chapter 8 demonstrates how to build a chat meeting applet and introduces the Places Service.

Tutorial Conventions

The following font conventions are used in this tutorial:

- Sample code is in `Courier New`.
- Sample code that has been added to a previous sample step is in **`bold Courier New`**.
- New terms and emphasis are in *italics*.

Related Documents

IBM Sametime *Java Toolkit Developer's Guide*

IBM Sametime *Java Toolkit Javadoc Reference*

Additional Information

Additional information can be found at the following Web sites:

http://www.lotus.com/sametime

http://www.ibm.com/developerworks/lotus

Part I

Part I of this tutorial introduces Sametime and provides an overview of the Sametime Java Toolkit architecture and services. If you are already familiar with Sametime and the toolkit architecture, skip Part I and proceed to Part II of this tutorial.

Chapter 2 presents Sametime from an end-user perspective and describes the features available when enabling applications with Sametime. The reader is introduced to the *Community Services*. Developers familiar with Sametime can skip this chapter and start from Chapter 3.

Chapter 3 provides an overview of the toolkit architecture and services. The chapter explains how to run the samples in this tutorial and provides a "Hello World" sample for logging in to the Sametime server with the toolkit.

Chapter 2. Introducing Sametime

Overview

This chapter provides a brief overview of the core services provided by Sametime, including awareness and chat. Applications enabled with Sametime provide the user with these real-time collaborative capabilities via the Community Services.

Once users are logged on to the Sametime server, they can see who is online, communicate, and work together in instant or scheduled meetings.

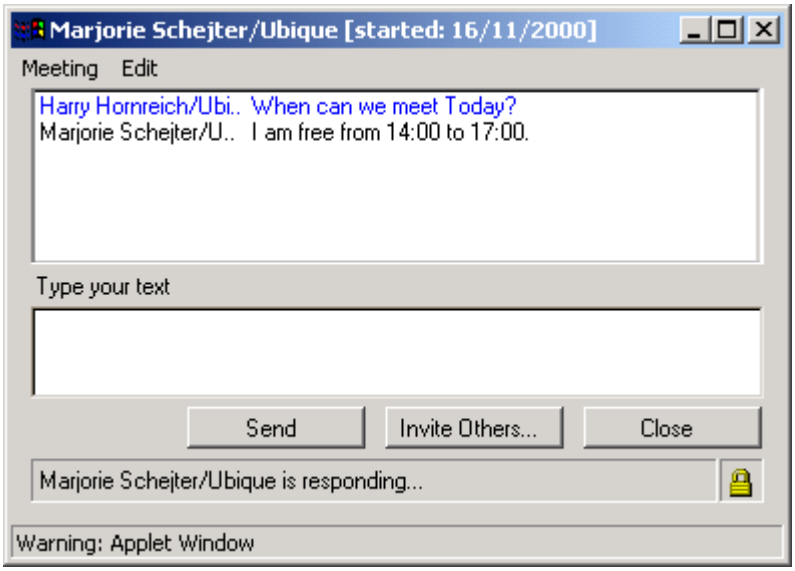
Additional information about Sametime is available at http://www.lotus.com/sametime

Sametime Community Services

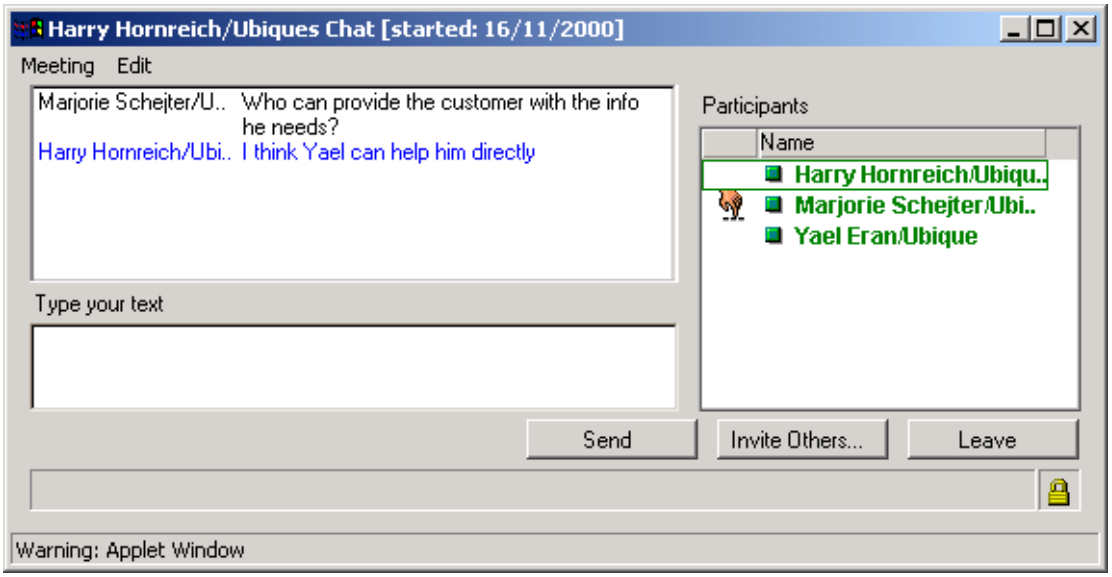
Sametime Community Services provide key community functionality, such as People Awareness, Places, Place-Based Awareness, Instant Messaging, and Chat.

Chat

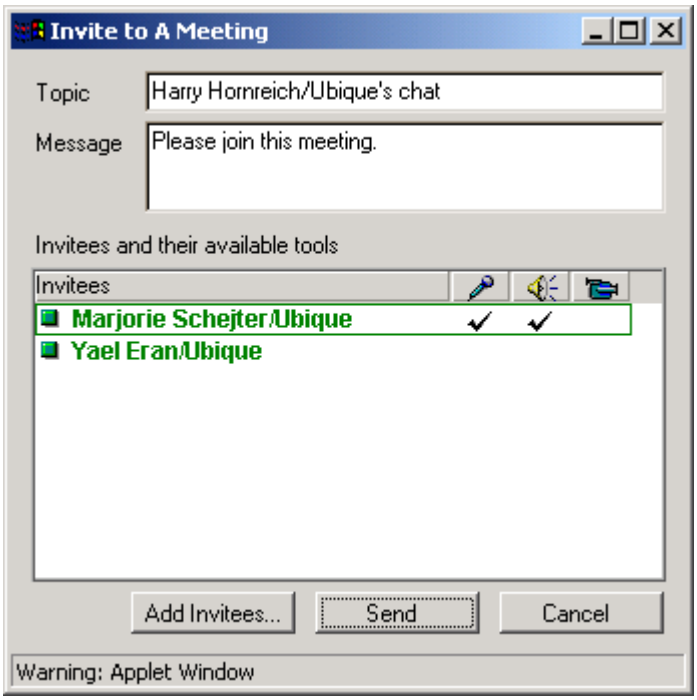
"Chat" refers to the exchange of text messages between online meeting participants. The following chat window is displayed when two users exchange instant messages. Participants can invite others to join, creating a chat with more than two participants (an "N-way chat"). Below is an example of a Chat window.



The following window is displayed when more than two people are exchanging instant messages.

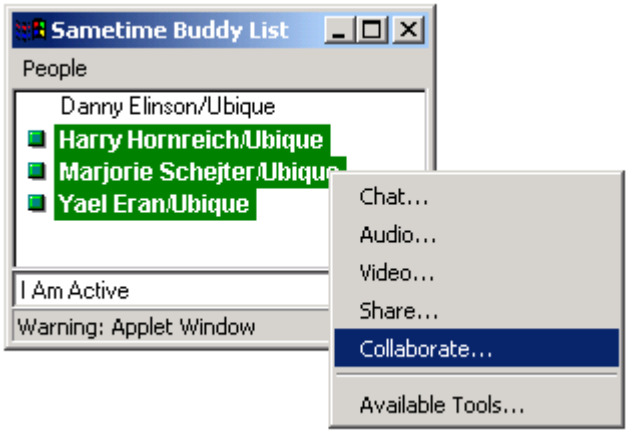


Any participant can open the "Invite to a Meeting" dialog to invite other online users to join an ongoing chat. This dialog is used to create an invitation for others to join the chat. The participant defines a list of invitees and an invitation message in this dialog:



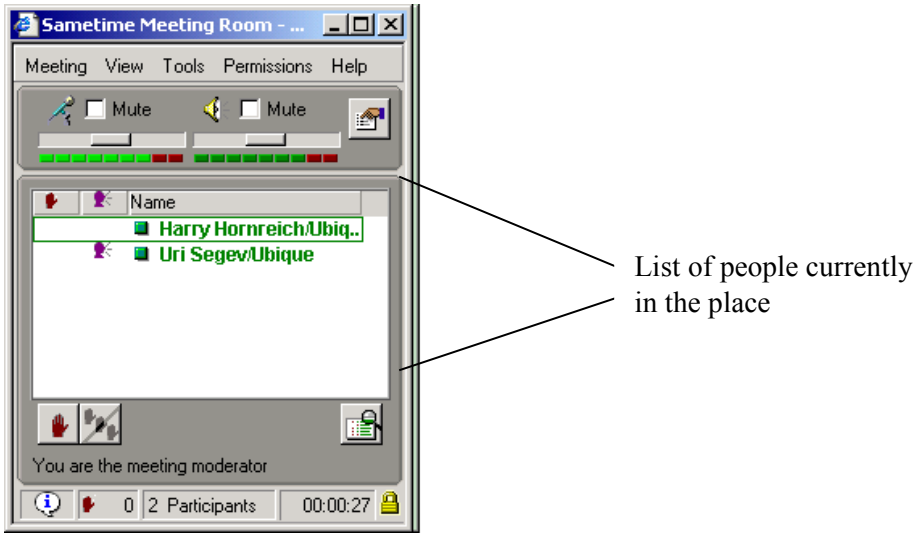
People Awareness

People Awareness allows users to see the online status of a pre-defined list of users. For example, this list could be the list of the sender and e-mail addresses. Any online users whose names appear in green text in the list are available to invite to an instant meeting. Create the instant meeting by selecting one or more “green names” and selecting the appropriate meeting type from the shortcut menu.



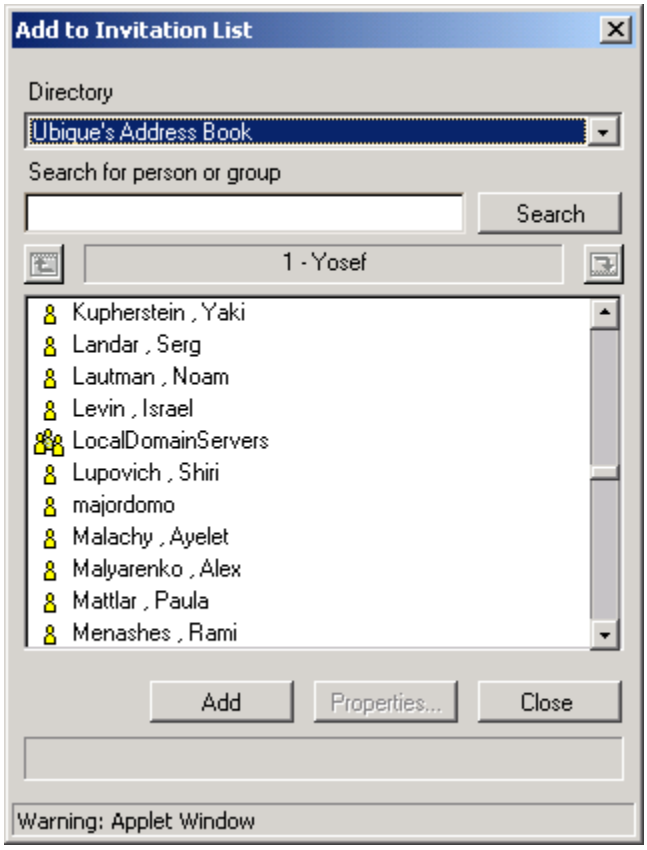
Place-Based Awareness

Place-Based Awareness allows users to see the online status of users who are currently in a virtual place. A virtual place can be a particular discussion database or an online meeting. Below is an example of an online meeting containing a list of people currently in the meeting.



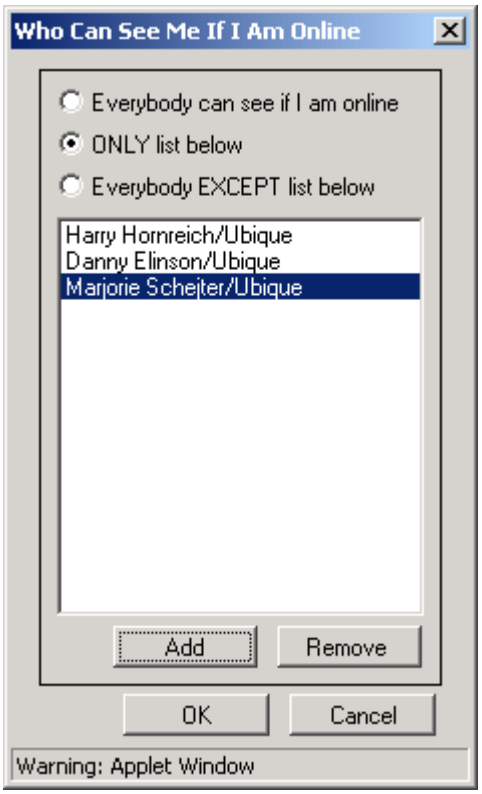
Sametime Directory Browsing

Sametime Directory Browsing provides a view of the available directories in the Sametime community and their contents. With Sametime Directory Browsing you can browse and search for users and groups within a directory and obtain group content information, such as a list of the members in the group. Below is an example of Sametime Directory Browsing:



Sametime Privacy

Sametime Privacy allows users to set their privacy settings. These settings determine who can see you while you are logged in to the community. (Note that you will not be able to see those who you decide cannot see you.) You can view and modify these privacy settings. Below is an example of the privacy setting window:



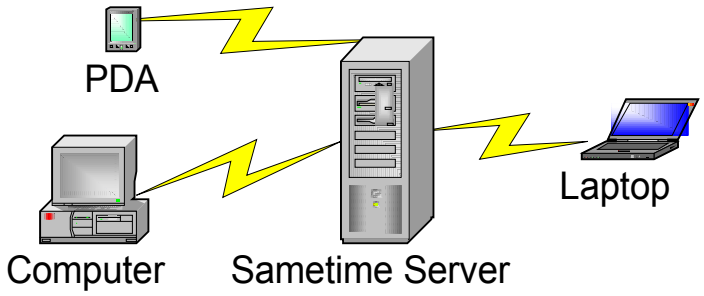
Chapter 3. Getting Started with the Toolkit

The latest release of the Sametime Java Toolkit includes documentation revisions and code fixes. For a summary of what's new, see the Readme.

If you are developing a Web site, you might also want to consider using the Sametime Links Toolkit. The Sametime Links Toolkit can be found in the client/stlinks directory in the Sametime SDK.

Sametime Architecture

Sametime has a client-server architecture. Sametime Connect and the Meeting Room Client (MRC) are examples of clients that are available with Sametime. The Sametime server makes no distinction between applets and applications that have been Sametime-enabled. All client-to-client communication such as instant messaging passes through the Sametime server. Therefore, once the client is logged in to the Sametime server, it has access to all Sametime services and can communicate with any other Sametime client logged in to the Sametime server.

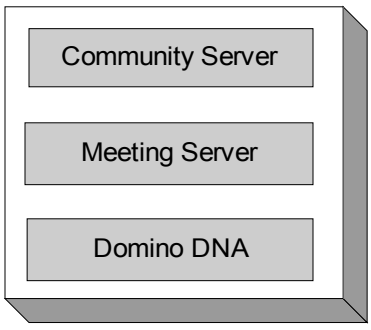


The Sametime server consists of three servers that interact:

The **Community Server** – Provides all Sametime Community Services such as login and awareness.

The **Meeting Server** – Provides all Sametime Meeting Services such as screen sharing and IP audio and video. (Note that IP audio and video require installation of the Multimedia Services package.)

The **Domino™ DNA –** Provides core Sametime services such as directory access, authentication, and the HTTP server.



Sametime Server

This toolkit has been designed to correspond to the architecture of the Sametime server, and to provide access to the various Community and Meeting Services.

The Sametime Java Toolkit

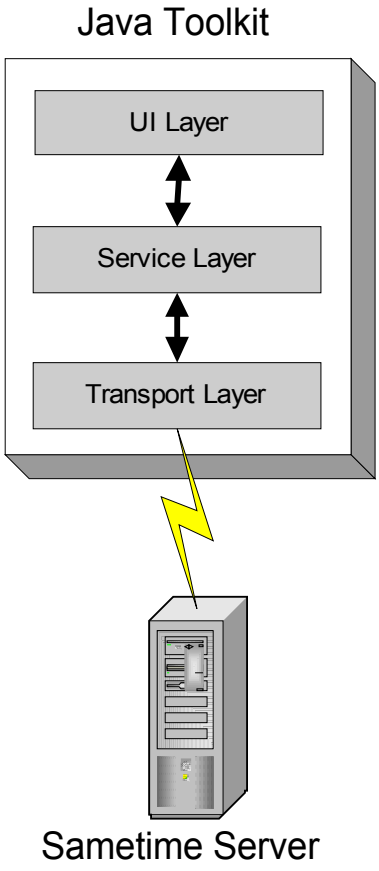
The Sametime Java Toolkit is powerful, yet simple to use. It provides access to core Sametime services, such as awareness and instant messaging.

The toolkit has a layered architecture with three layers:

Transport Layer – All communication with the Sametime server passes through this layer.

Service Layer – This layer provides access to the Sametime Community Services.

UI Layer – This layer provides UI components and Sametime-enabled AWT components **NOTE:** The UI Layer has been deprecated starting with Sametime 7.5.



The toolkit is modular, thread-safe, and extendable, providing an object-oriented API that exposes the entire feature set provided by Sametime.

Modular

The Sametime Java Toolkit is component-based, and the different services it provides are divided among many components. The developer has complete control over which components are loaded, based on the Sametime services required for the application. The developer writes the code so that only the components providing these services are loaded. When the application is delivered, only the Java packages containing the required components from the toolkit need be included. This modular structure results in smaller deliverables and shorter

download time for the end-user. See Appendix B in the *Sametime Java Toolkit Developer's Guide* for additional information.

Thread-safe

The Sametime Java Toolkit is thread-safe, so that a Sametime-enabled application built on top of the toolkit can have many threads. Calls to the toolkit API can come from different threads; the toolkit handles multiple threads without the developer needing to do any additional work.

Extendable

In the Sametime Java Toolkit, developers can add components that provide new services and a new user interface (UI), making the toolkit extendable by both core Sametime developers and by third-party developers. This tutorial does not describe how to write such components.

Object-oriented API

The Sametime Java Toolkit provides a natural object-oriented API for the Java developer. In previous toolkit versions, each service was a class with a set of methods. In the Sametime Java Toolkit, each service is an interface implemented by a component class. Many services create, and require the use of, one or more additional helper objects in order to execute the service functionality. This programming model is familiar to Java developers. It fits easily into existing Java code and programming paradigms.

The Toolkit Services

This section describes the services provided by the Sametime Java Toolkit. Note that the Community Services belong to the service layer of the architecture. They therefore provide no UI, only programmatic access to the different Sametime services.

Community Services

The following table lists and describes the Community Services available in the Sametime Java Toolkit.

Table listing available services in Sametime Java Toolkit.

Name of Service	Description
Community Service	This service is a required component in almost any use of the toolkit. It allows you to log in, log out, and make changes to your online status, online attributes, and privacy settings.
Announcement Service	This service provides the ability to send and receive announcements between Sametime users.
Awareness Service	This service provides notifications of changes in the online status and attributes of users in the community. This service is sometimes referred to as "People Awareness."
Buddy List Service	This service provides the ability to get and set the user contact list stored on the Sametime server and provides conversion methods from String to BL object and vice versa, without having to deal with the low level protocol as defined by the storage service. Using this service ensures the integrity and compatibility of the contact list data.
File Transfer Service	This service provides the ability to send and receive files between Sametime users.
Instant Messaging Service	This service provides one-to-one communication between clients. It is used mostly for Instant Messaging chat between users, but it can also be used to exchange any kind of text or binary data.
Places Service	This service provides the ability to create virtual meeting places that users can enter or leave, see who is in the meeting place, and share activities. This service is sometimes referred to as "Place-Based Awareness." It is a cornerstone of the Sametime architecture.
Storage Service	This service provides server-side storage of user-related data. A user can access his personal data from any Sametime-enabled application. For example, Sametime Connect uses this service to store a user's contact list on the Sametime server so that the user can access it regardless from where he logs in.
Lookup Service	This service provides name resolving and group content lookup.
Directory Service	This service provides directory-browsing services.
Post Service	This service allows posting a message to many users at once.
Token Service	This service allows generation of temporary login tokens.
Names Service	This service provides names and nicknames services.
Multicast Service	This service provides the ability to send messages to multiple recipients.

Running the Tutorial Samples

You can run the tutorial samples directly from the toolkit pages; you do not need to enter or compile and code. Make sure the toolkit is in \<sametime_server_data_folder>\domino\html\, and access it with http://sametime-hostname/ STJavaSamples/index.html Then do the following:

1. Make sure you are registered on the Sametime server and have a user name and password. If not, follow the "Register" link on the Sametime server home page to register.

2. Note that some of the samples require that the user names "Tom", "Dick", and "Harriet" be registered in the community. You might want to log in as these users using Sametime Connect to see how the samples track the online status of these users.

3. Click the "Samples" link on the toolkit home page (shown above).

4. Select the sample you want to run from the list of samples. (The tutorial samples are listed in the order they appear in the tutorial.) You can view the sample sources directly from this page by clicking the appropriate icon next to the samples name.

5. Enter your user name and password in the login form.

You are now on a page that describes and runs the sample you requested.

Downloading the Sample Code

If you prefer to run the samples yourself or change the sample code, all the sample code is included in the STJavaSamples.zip in the download directory of the Java Toolkit.

1. Compile the Java source code of the sample you want to run using any JDK 1.4.2 compatible development environment. Be sure to add the necessary libraries from the Java Toolkit bin directory (CommRes.jar and stjavatk.jar or STComm.jar) to your build path.

2. If the sample is an applet, modify the sample HTML file by replacing the user name and password in the file with your user name and password. All samples are preconfigured to log in as the user "Tom" with the password "sametime." If you register this user on the Sametime server, you will not need to modify the sample HTML files.

3. Run the sample.

If the sample is an applet, run it using Microsoft® Internet Explorer, Netscape, or the standard applet viewer utility provided by Sun Microsystems. If the sample is a Java application, run it using the Java development environment.

Hello World Sample

It is traditional programming practice to start any new language description with a simple "Hello World" sample. Because this tutorial introduces the Sametime platform and programming model (and not a new language), the "Hello World" sample for this tutorial will run a simple applet that logs in to a Sametime server and displays the message "Hello Sametime."

The following is the Hello World applet HTML and source code. For instructions on how to run this sample, see the section "____" in Chapter 3.

Hello World HTML Code

The following is the HTML code for the Hello World Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime Hello World Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=HelloWorldApplet.class NAME=HelloWorldApplet
            WIDTH=95% HEIGHT=95%>
<PARAM NAME='archive' VALUE='../CommRes.jar,../STComm.jar'>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
</APPLET>
</BODY>
</HTML>
```

Hello World Source Code

The following is the source code for the Hello World Sample:

HelloWorldApplet.java

```
import java.awt.*;
import java.applet.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.community.*;

/**
 * Sample Sametime Hello World applet.
 */
public class HelloWorldApplet extends Applet
{
  private STSession m_session;
  private CommunityService m_comm;

  /**
   * Applet initialized. Create the session, load all components,
```

```
 * start the session and then login.
 */
public void init()
{
  try
  {
    m_session = new STSession("HelloWorldApplet " + this);
    m_session.loadAllComponents();
    m_session.start();
    m_comm = (CommunityService)
              m_session.getCompApi(CommunityService.COMP_NAME);
    m_comm.loginByPassword(getCodeBase().getHost().toString(),
                           getParameter("loginName"),
                           getParameter("password"));
}
  catch(DuplicateObjectException e)
  {
    e.printStackTrace();
  }
}

/**
 * Prints "Hello Sametime" to the applet area.
 */
public void paint(Graphics g)
{
  Font f = new Font(g.getFont().getName(), Font.BOLD, 20);
  g.setFont(f);
  g.setColor(Color.black);
  g.drawString("Hello Sametime", 30, 30);
}

/**
 * Applet destroyed. Logout, stop and unload the session.
 */
public void destroy()
{
  m_comm.logout();
  m_session.stop();
  m_session.unloadSession();
```

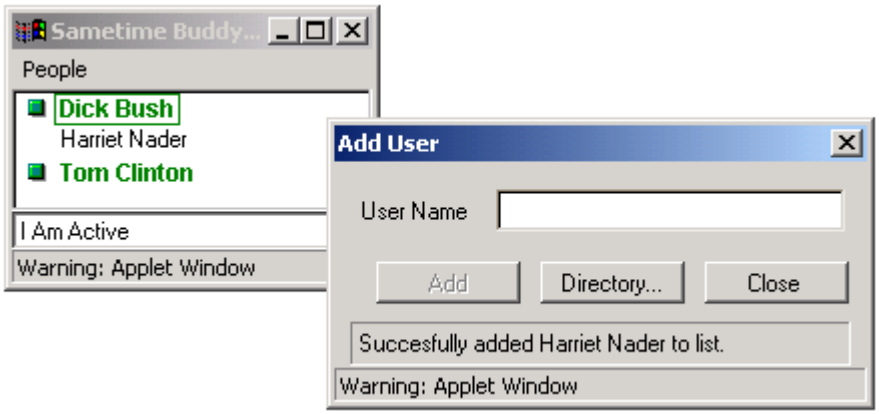
```
    }
}
```

In this simple applet, no notification is received on whether the login was successful. In order to check that the applet has actually logged in, run Sametime Connect using a different user name than the one you used in the sample. Add the user you used in the sample to your Sametime Connect contact list. You should see the sample user name online when the sample is run and offline when the applet is destroyed. Try sending an instant message to the applet user.

Part II of this tutorial provides a full explanation of the Hello World sample code and describes how you can receive login notifications.

Part II

Part II introduces the Sametime Java Toolkit and builds a simple Sametime Buddy List applet.



The Buddy List applet includes features that allow you to:

- Log in to Sametime.
- Add users to the contact list by name or by browsing the directory.
- Store the contact list on the Sametime server.
- Start instant messages, audio, video, share, and collaboration meetings with one or more users in the list.
- Receive instant messages and invitations to any type of Sametime meeting.
- Change your online status in the community.
- Change your privacy settings.

This section provides step-by-step instructions for building the Buddy List applet and adding the above features to it. Part II also explains how to use many of the Community Services in the toolkit.

Chapter 4. Login Sample

Overview

This chapter presents and explains a Login applet that expands on the Sametime Hello World applet by adding login event notifications. These notifications inform users if they have successfully logged in to Sametime.

This chapter also includes a full explanation of the entire Hello World applet code.

The Login applet logs in to the Sametime server once it is initialized and logs out when the applet is destroyed. The applet has successfully logged in to the Sametime server if the "Logged In" text is seen on the applet area. You can also add the user "Tom" (who will be used in all the samples) to your Sametime Connect contact list and log on using your Sametime login. You should see the user "Tom" become green, which indicates that "Tom" is online, once the applet is run. Try sending an instant message to "Tom". You will see that this simple applet already contains a lot of Sametime functionality due to the component architecture of the toolkit.

The following is the HTML and Java source code for Login Sample. The remainder of the chapter provides a detailed explanation of the code.

For instructions on how to run this sample, see the section "____" in Chapter 3.

Login HTML Code

The following is the HTML code for the Login Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime Login Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=LoginApplet.class NAME=LoginApplet WIDTH=95% HEIGHT=95%>
<PARAM NAME='archive' VALUE='../CommRes.jar,../STComm.jar'>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
</APPLET>
</BODY>
</HTML>
```

Login Source Code

The following is the source code for the Login Sample:

LoginApplet.java

```
import java.awt.*;
import java.applet.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.community.*;

/**
 * Sample applet that logs in to a Sametime server.
 */
public class LoginApplet extends Applet implements LoginListener
{
  private STSession m_session;
  private CommunityService m_comm;
  private String m_currentStatus = "Logged Out";

  /**
   * Applet initialized. Create the session, load all components,
   * start the session and then login.
   */
  public void init()
  {
    try
    {
      m_session = new STSession("LoginApplet " + this);
      m_session.loadAllComponents();
      m_session.start();
      login();
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }
```

```
}

/**
 * Login to the community using the user name and password
 * parameters from the html.
 */
private void login()
{
  m_comm = (CommunityService)
      m_session.getCompApi(CommunityService.COMP_NAME);
  m_comm.addLoginListener(this);
  m_comm.loginByPassword(getCodeBase().getHost().toString(),
                         getParameter("loginName"),
                         getParameter("password"));
}

/**
 * Prints a simple connectivity status message.
 */
public void paint(Graphics g)
{
  Font f = new Font(g.getFont().getName(), Font.BOLD, 20);
  g.setFont(f);
  g.setColor(Color.black);
  g.drawString(m_currentStatus, 30, 30);
}
/**
 * Logged in event. Print logged in msg to console & applet.
 */
public void loggedIn(LoginEvent event)
{
  System.out.println("Logged In");
  m_currentStatus = "Logged In";
  repaint();
}

/**
 * Logged out event. Print logged out msg to console & applet.
 * Leave default behavior which will display a dialog box.
 */
```

```
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");
    m_currentStatus = "Logged Out";
    repaint();
  }

  /**
   * Applet destroyed. Logout, stop and unload the session.
   */
  public void destroy()
  {
    m_comm.logout();
    m_session.stop();
    m_session.unloadSession();
  }
}
```

STSession and Components

Most Sametime-enabled applications contain code involving `STSession` and its components. The first Sametime code in the applet's init() method is the following:

```
m_session = new STSession("LoginApplet " + this);
m_session.loadAllComponents();
m_session.start();
```

Sametime functionality is split among several components, each responsible for providing a service. For example, the Instant Messaging Service is responsible for sending and receiving instant messages to and from remote partners. The Awareness Service provides notifications about the online status and attributes of users in the community.

Creating a New STSession Object

A STSession object holds a set of components and allows access to them. Usually, each Sametime-enabled application has only one session object. However, advanced applications can use several sessions or share a single session between several applets. Because some applications use more than one session object, every STSession object has a name that can be found in a static session table[1].

The following code creates a new STSession object with a unique name derived from the applet name:

```
m_session = new STSession("LoginApplet " + this);
```

[1] Use the class com.lotus.sametime.core.comparch.SessionTable to look up a session object by its name.

Because an exception is thrown if a session with the requested session name already exists in the session static table, we need to wrap the session creation code with an exception try/catch construct, as follows:

```
try
{
  m_session = new STSession("LoginApplet " + this);
  …
}
catch(DuplicateObjectException e)
{
  e.printStackTrace();
}
```

Adding Components to the STSession Object

After the session is created, add components to it:

```
m_session.loadAllComponents();
```

The above code loads all available components to the session[2]. Although loading all available components is not always necessary, it is the safest option if you are not entirely familiar with each component. As your understanding of the components increases, you will be able to select the specific components that you need. See Appendix B of the *Sametime Java Toolkit Developer's Guide* for more information on how to load specific toolkit components and how to package your final application with a subset of toolkit components.

Starting and Stopping the STSession Object

The session and the components must be started before they can be used. The following method requests the session to start itself and its loaded components:

```
m_session.start();
```

When cleaning up a Sametime-enabled application, stop and unload the session. The following code in the applet's destroy() method unloads the session:

```
m_session.stop();
m_session.unloadSession();
```

Login and Logout

After creating a session, loading the components, and starting them, the next step is to log in to the Sametime community. You must log in before you can use most of the services of the toolkit. Once you are logged in, other logged-in users can see that you are online and can receive notifications about your online status and attributes by using the Awareness Service.

Getting a Reference to the STBase Component

The STBase component implements the Community Service interface and is responsible for logging in and out of the Sametime community. Since the STBase component is loaded when all components are loaded to the session, you must get a reference to this component to be able to log in. The following code gets the necessary reference:

```
m_comm = (CommunityService)
         m_session.getCompApi(CommunityService.COMP_NAME);
```

[2] STSession tries to load all Toolkit components included in the final application package.

The STSession.getCompApi() method requests a reference to a component, based on its name. Every Sametime component has a COMP_NAME field used for this purpose. The name of the STBase component is CommunityService.COMP_NAME. The returned component reference needs to be cast to the right type, since STSession does not distinguish between components and does not know the type of the requested component.

Logging In and Out of Sametime

In the applet's login() method, the following code attempts to log in to the Sametime community using a user name and password. The user name and password are passed to the applet:

```
m_comm.loginByPassword(getCodeBase().getHost().toString(),
                       getParameter("loginName"),
                       getParameter("password"));
```

Note that this code assumes that the Login sample is installed on the Sametime server and retrieves the server name by using the statement:

```
getCodeBase().getHost().toString()
```

In the applet's destroy() method, the following code logs you out of the Sametime community before the session is stopped and unloaded:

```
m_comm.logout();
```

Receiving Events

When the login() method returns, you might not be logged in to the Sametime community. Because you are connecting to a remote community, the login process can take time. Initiate the connection to the server, send authentication data, and wait for confirmation. To avoid waiting for the confirmation, you can ask the Community Service to notify you when the login procedure is finished.

Adding a Login Listener

Using the standard Java event model, add a listener for login events as follows:

```
m_comm.addLoginListener(this);
```

This code alerts the Community Service that the applet wants to receive notifications about login events. For this code to compile, the applet needs to implement the LoginListener interface. Implementing this interface means that the applet must define two event handlers: loggedIn() and loggedOut(). The loggedIn() event handler is called when the login process completes successfully. Whatever must be done when we first log in should occur in this event handler.

Upon failing to log in to the Sametime community, you will receive the loggedOut() event with a reason code explaining the failure. In this applet, either "Logged In" or "Logged Out" will be printed to the applet area when either of these events is received:

```
/**
 * Prints a simple connectivity status message.
 */
public void paint(Graphics g)
{
  Font f = new Font(g.getFont().getName(), Font.BOLD, 20);
  g.setFont(f);
  g.setColor(Color.black);
  g.drawString(m_currentStatus, 30, 30);
}

/**
```

```
   * Logged in event. Print logged in msg to console & applet.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Logged In");
    m_currentStatus = "Logged In";
    repaint();
  }

  /**
   * Logged out event. Print logged out msg to console & applet.
   * Leave default behavior which will display a dialog box.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");
    m_currentStatus = "Logged Out";
    repaint();
  }
```

Note that you will also receive the loggedOut() event if you are logged in to Sametime and you are disconnected from the community.

Using Other Toolkit Listeners

The same technique for receiving events is used in all of the toolkit's services. The Sametime-enabled application implements a listener interface of events it is interested in and registers it with the appropriate service. It then issues requests by calling methods in that service. The service lets the application know that something has happened by sending an event to the listener implementation.

You can register more than one listener to receive events from a service. The registered listeners are called sequentially in this case.

UI Components

If you experiment with the sample login applet, you will notice some interesting behavior. If the applet fails to log in for some reason, such as the network is down or the password is incorrect, an error message displays. Also, if you log in using the Sametime Connect client with a user name and password different from the one you used in the Login sample, add the sample user to your contact list and then send him a message when the applet is running, an Instant Message (IM) window will display in the applet. Both the error message and the IM window are the work of UI components that were loaded to the applet session when it called the loadAllComponents() method.

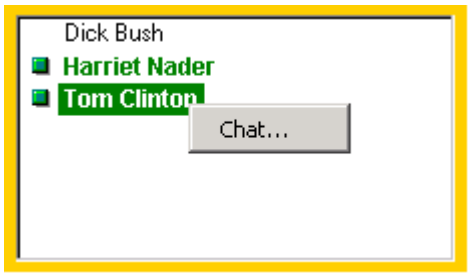
ChatUI is the UI component responsible for displaying and handling IM and chat windows. It listens to events from the Instant Messaging Service component. Whenever someone sends the user a message, it displays a window to let the user know that someone is trying to contact him. Likewise, the CommUI component provides popup error messages when the applet disconnects or logs out from the community. This rich functionality is available without any extra effort on the developers' part due to the toolkit component architecture.

Chapter 5. Live Names Sample

Overview

In this Live Names sample we add UI to display the online status of a list of Sametime users. To accomplish this task an AwarenessList is used. AwarenessList is one of the most useful Sametime-enabled AWT components provided by the toolkit.

The AwarenessList AWT component displays the list of Sametime users in black if they are offline and in green if they are online. Statuses of the users are shown with the appropriate icons. Right-click on one name to start an instant message or on multiple names to start a chat meeting. As shown below:



The Live Names Sample also introduces the Lookup Service and name resolving. Name resolving is the process in which user names are transformed into Sametime user objects. Each user object represents a unique user in the community. Most of the new code in this applet (shown in bold) deals with name resolving.

For instructions on how to run this sample, see the section "_____" in Chapter 3.

Live Names HTML Code

The following is the HTML code for the Live Names Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime LiveNames Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=LiveNamesApplet.class NAME=LiveNamesApplet
           WIDTH=95% HEIGHT=95%>
<PARAM NAME='archive' VALUE='../CommRes.jar,../STComm.jar'>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
<PARAM NAME='watchedNames' VALUE='Tom,Dick,Harriet'>
</APPLET>
</BODY>
</HTML>
```

Live Names Source Code

The following is the source code for the Live Names Sample:

LiveNamesApplet.java

```
import java.awt.*;
import java.applet.*;
import java.util.*;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.STUser;
import com.lotus.sametime.community.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.awarenessui.list.AwarenessList;

/**
 * Sample applet that displays a list of live user names.
 */
public class LiveNamesApplet extends Applet
    implements LoginListener, ResolveListener
{
  private STSession m_session;
  private CommunityService m_comm;
  private AwarenessList m_awarenessList;

  /**
   * Applet initialized. Create the session, load all components,
   * start the session and then login.
   */
  public void init()
  {
    try
    {
      m_session = new STSession("LiveNamesApplet " + this);
      m_session.loadAllComponents();
      m_session.start();

      setLayout(new BorderLayout());
```

```
      m_awarenessList = new AwarenessList(m_session, true);
      add(m_awarenessList, BorderLayout.CENTER);

      login();
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    m_comm = (CommunityService)
        m_session.getCompApi(CommunityService.COMP_NAME);
    m_comm.addLoginListener(this);
    m_comm.loginByPassword(getCodeBase().getHost(),
                           getParameter("loginName"),
                           getParameter("password"));
  }

  /**
   * Logged in event. Print logged in msg to console.
   * Resolve the users to add to the awareness list.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Logged In");

    m_awarenessList.addUser(
                    (STUser)event.getLogin().getMyUserInstance());

    LookupService lookup = (LookupService)
      m_session.getCompApi(LookupService.COMP_NAME);

    Resolver resolver =
```

```
      lookup.createResolver(false,  // Return all matches.
                            false,  // Non-exhaustive lookup.
                            true,   // Return resolved users.
                            false); // Do not return resolved groups.

  resolver.addResolveListener(this);

  String[] userNames = getUserNames();
  resolver.resolve(userNames);
}

/**
 * Helper method to read a list of user names from the html
 * parameter 'watchedNames'.
 */
String[] getUserNames()
{
  String users = getParameter("watchedNames");

  StringTokenizer tokenizer = new StringTokenizer(users, ",");
  String[] userNames = new String[tokenizer.countTokens()];

  int i = 0;
  while(tokenizer.hasMoreTokens())
  {
    userNames[i++] = tokenizer.nextToken();
  }
  return userNames;
}

/**
 * Users resolved succesfully event. An event will be generated for
 * each resolved user. Add the resolved user to the awareness list.
 */
public void resolved(ResolveEvent event)
{
  m_awarenessList.addUser((STUser) event.getResolved());
}

/**
```

```
    * Handle a resolve conflict event. Will be received in the case
    * that more then one match was found for a specified user name.
    * Add the users to the awareness list anyway.
    */
  public void resolveConflict(ResolveEvent event)
  {
    STUser[] users = (STUser[]) event.getResolvedList();
    m_awarenessList.addUsers(users);
  }

  /**
   * Resolve failed. No users are available to add to the list.
   */
  public void resolveFailed(ResolveEvent event)
  {
  }

  /**
   * Logged out event. Print logged out msg to console. Leave default
   * behavior which will display a dialog box.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");
  }

  /**
   * Applet destroyed. Logout, stop and unload the session.
   */
  public void destroy()
  {
    m_comm.logout();
    m_session.stop();
    m_session.unloadSession();
  }
}
```

Awareness List

An AwarenessList is an AWT component provided by the toolkit. Since it inherits from the Component class, it can be added to any AWT Container, making the AwarenessList simple to use.

To review the Live Names source code, start with the init() method. You are already familiar with the code discussed previously, which initializes the session object. The following code has been added to this method:

```
setLayout(new BorderLayout());
m_awarenessList = new AwarenessList(m_session, true);
add(m_awarenessList, BorderLayout.CENTER);
login();
```

The first line of code chooses a layout manager in a manner familiar to all Java programmers. The next two lines of code create the awareness list and add it to the applet layout manager. The last statement calls the login() helper method to log in to Sametime. Note that the session object is passed to the AwarenessList constructor, which binds the awareness list to the session and allows the list to be Sametime-enabled.

When an AwarenessList is created, it is empty by default. Before users can be added to the awareness list, their names must be resolved. Resolving a user name means searching for a user in the community directory, and if found, converting that name to an object of type STUser. An STUser object uniquely identifies a Sametime user. Most Sametime services that handle users accept an STUser object, but not a name, because names are not unique in Sametime. The in-depth discussion of the Sametime user model at the end of this chapter explains why user names are not unique in Sametime.

In this sample, we have also shown how you can add yourself to the AwarenessList with the following code in the loggedIn() event:

```
m_awarenessList.addUser(
                    (STUser)event.getLogin().getMyUserInstance());
```

Name Resolving

Resolve operations are performed using the Lookup Service. You must be logged in before you can resolve names; therefore, the resolve request code was included in the loggedIn() event handler.

Creating a Resolver Object

Before resolving a name, a reference to the Lookup Service component is needed. This task is accomplished through the session object:

```
LookupService lookup = (LookupService)
  m_session.getCompApi(LookupService.COMP_NAME);
```

Next, request the Lookup Service to create a Resolver object, which is the object that will resolve the different names. There are several options for resolving names, and these options are specified when the Resolver object is created from the Lookup Service. Once created, the resolve options cannot be changed for the complete lifecycle of the Resolver object. Available resolve options are:

- OnlyUnique – This option determines whether the created resolver object should resolve successfully only if the provided name is matched exactly in the Sametime directory.

- ExhaustiveLookup – This option determines whether the created resolver object should perform an exhaustive lookup through all directories or stop in the first directory where a match is found.

- ResolveUsers – This option determines whether the created resolver should search the directory for users matching the provided name.

- ResolveGroups – This option determines whether the created resolver should search the directory for groups matching the provided name.

Note that resolving can be done on user names only, group names only, or both user and group names. In this sample, the resolver is requested to return all possible matches, not to look through multiple directories, and to resolve user entries but not group entries:

```
Resolver resolver =
    lookup.createResolver(false,  // Return all matches.
                          false,  // Non-exhaustive lookup.
                          true,   // Return resolved users.
                          false); // Do not return resolved groups.
```

Resolving Names

In the final lines of the login event handler, the applet is added as a listener to the resolver events so you are notified of the resolve results when they arrive. Use a helper method to parse the user names from the watchedNames parameter of the applet. Finally, call the resolve() method of the resolver to resolve these names:

```
resolver.addResolveListener(this);
String[] userNames = getUserNames();
resolver.resolve(userNames);
```

ResolveListener Events

When an array of names in a request is resolved, the resolve operation can succeed for some names and fail for other names. For each successfully resolved name, the resolved() event handler is called separately.

The resolved() Event

If the name is resolved successfully, the returned STUser object is added to the awareness list with the following code:

```
public void resolved(ResolveEvent event)
{
  m_awarenessList.addUser((STUser) event.getResolved());
}
```

Once addUser() is called, the awareness list adds the user to an Awareness Service WatchList object and listens to the appropriate events from this object. The user will immediately appear on the list, and his online status will be tracked from that moment until the user is removed from the list.

The resolveConflict() Event

As previously mentioned, a resolve operation can fail. In one possibility, the Sametime directory could contain several entries that match the requested name. In this case, a resolveConflict() event is generated, and the list of possible matches is available from the `event` object. In our Live Names sample, all matched users are added to the awareness list as follows:

```
public void resolveConflict(ResolveEvent event)
{
  STUser[] users = (STUser[]) event.getResolvedList();
  m_awarenessList.addUsers(users);
```

```
    }
```

A real application would probably display all matches to the user running the application and let the user choose the correct match[3]. In Chapter 7 of this tutorial another way of handling resolve conflicts will be shown.

The resolveFailed() Event

The resolve operation might fail completely because no user or group matches the resolve name according to the resolver options. In the event of a failure, the resolveFailed() event is generated. Use the getReason() method of the passed `event` object to return a reason code providing more information about why the resolve operation failed.

Every Sametime event that is generated because of a failure has a reason code field. The STError class, in the com.lotus.sametime.core.constants package, contains all possible values of this reason code field. For example, the reason code STError.ST_ERROR_VPRESOLVE_NO_MATCHES indicates that the resolve operation failed because no match was found in the directory. In this sample nothing is done in the case of a resolve failure. A real application would probably display a message box letting the user know why a resolve operation failed.

Sametime User Model

This advanced section describes the Sametime User Model in detail.

User IDs

The Sametime user model is flexible and allows Sametime to be integrated with different kinds of user directories such as Domino, LDAP, and others. Sametime requires that every user in the community have a unique identifier called a *user ID*. The user ID is a string that uniquely identifies the user internally in Sametime but is never actually shown to the user. It is the responsibility of the administrator to ensure that any new user added to the system has a unique user ID.

Apart from the required user ID, Sametime is very flexible with how users can log in and with what name they are displayed in the community. The following table displays possible user entries in the Sametime directory:

Table listing possible user entries in the Sametime directory.

	Entry 1	**Entry 2**	**Entry 3**	**Entry 4**
User ID	UID1	UID1	UID2	UID3
Login Name	mickey	mick	daisy	popeye
Password	donald	goofy	flowers	spinach
User Name	Mickey	Mickey Mouse	Mickey Mouse	Sailorman

In this table there are three unique user entries: UID1, UID2 and UID3. UID1 can log in using the combination mickey/donald or the combination mick/goofy as his login name and password. In either case, Sametime will authenticate the request as UID1 (by looking through the table) and will make him known with the name “Mickey” for the first combination and “Mickey Mouse” for the second combination.

---

[3] The Toolkit actually provides a UI to allow the user to make such a selection. See the class com.lotus.sametime.commui.ResolvePanel.

A few points to note:

- The login name/password combination needs to be unique in the table and match an entry in order to log in to Sametime.
- The login name is used only to log in and has no other use after login is complete.
- A user can potentially appear with different user names in the same community. It is the administrator's responsibility to ensure that user names make sense to the end users.

In the Sametime user model a user can have several user names; and different users can have the same user name. In the table, UID1 and UID2 can both be logged in at the same time and have the same user name ("Mickey Mouse") from the point of view of all other users in the community. Although Sametime can determine that the same name represents two different users, the users in the community cannot distinguish between the names. The administrator must make sure such occurrences are avoided if they do not make sense in the community.

The STUser class encapsulates the notion of a unique user entry in the Sametime directory. Resolving the name "Sailorman" will return an STUser object representing UID3 with the user name "Sailorman." Sametime also supports the notion of directory-defined groups similar to e-mail groups. These groups also have a unique group ID and a non-unique group name. The `STGroup` class encapsulates the notion of a unique group entry in the Sametime directory.

To summarize the resolve process, Sametime resolves names by looking through all its directory entries to find matches.

Anonymous Logins

The ability to log in anonymously to a Sametime community allows users to request to log in without authenticating against the directory and to request a specific user name by which they will be known. The administrator can control this option and disable it if necessary. In addition, Sametime can decide to reject a certain user name for security reasons and provide a name such as "guest23" that indicates that this user has logged in anonymously.

Anonymous login is especially important for Sametime communities that need to allow access to the community to users outside of the community. A good example is a Customer Relationship Management (CRM) application that uses instant messaging to facilitate customer/support interaction. The support agents are registered in the community, but the external customers who want to log in should probably log in anonymously.

Anonymous users have additional restrictions on what operations they are allowed to perform in the community,and the administrator can grant or revoke these permissions as needed.

Login IDs

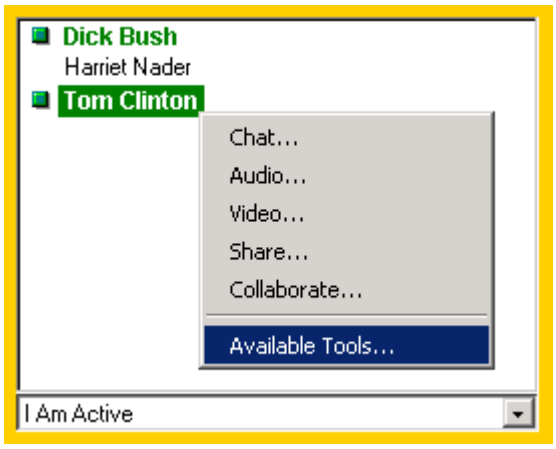
So far we have discussed the *static user model* of Sametime that is related to the authentication and resolve processes. The Sametime *run-time user model* is another related user model. Like the static user model, the Sametime run-time user model is flexible. A user can be logged in with more than one client at a single point of time to a single Sametime community. For example, a user can be logged in from Sametime Connect and from an MRC meeting at the same time. Both clients are unrelated and have separate physical connections to the Sametime server.

How will the Sametime server differentiate the different logins/connections if they belong to the same user? Each client that logs in to Sametime is provided with a unique *login ID* that is valid only for the current login session. Login IDs are guaranteed to be unique in run-time, which means that no two clients connected to a Sametime community at the same time will have the same login ID. In the above example, Sametime Connect and MRC will have the same user ID but different login IDs.

Chapter 6. Extended Live Names Sample

Overview

In the previous chapter, the Live Names Sample applet displayed a list of users with their online status and allowed users to receive and initiate instant messages and receive and initiate invitations to chat meetings.



In this chapter we will add two important features to our Live Names sample:

- The ability to change your online status – Allows you to control how other users see you when you are online. When you run the Live Names sample, you are aware of the online status of the users in your list, and anyone in the community can see your online status.

- The ability to initiate and join all types of meetings with online coworkers – Connects the sample to the Meeting Room Client (MRC) that is used for all meetings that have more features than just chat.

The MRC itself is an applet that is run inside a browser window. The Java Toolkit does not have any code to launch a browser since doing so is dependent on the context in which the toolkit is used. Therefore, by default the ChatUI component does not enable invitations to an MRC meeting or the initiation of such meetings. In this chapter, you will learn how to fill in the browser launching code and add this feature to any Sametime-enabled application.

For instructions on how to run this sample, see the section "_____" in Chapter 3.

Extended Live Names HTML Code

The following is the HTML code for the Extended Live Names Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime ExtLiveNames Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=ExtLiveNamesApplet.class NAME=ExtLiveNamesApplet
```

```
            WIDTH=95% HEIGHT=95%>
<PARAM NAME='archive' VALUE='../CommRes.jar,../STComm.jar'>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
<PARAM NAME='watchedNames' VALUE='Tom,Dick,Harriet'>
</APPLET>
</BODY>
</HTML>
```

Extended Live Names Source Code

The following is the source code for the Extended Live Names Sample:

ExtLiveNamesApplet.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.util.*;
import java.net.URL;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.types.STUser;
import com.lotus.sametime.core.types.STUserStatus;
import com.lotus.sametime.community.*;
import com.lotus.sametime.lookup.*;
import com.lotus.sametime.awarenessui.list.AwarenessList;
import com.lotus.sametime.awarenessui.av.AVController;
import com.lotus.sametime.chatui.ChatUI;
import com.lotus.sametime.chatui.MeetingListener;
import com.lotus.sametime.chatui.MeetingInfo;

/**
 * Sample applet that displays a list of live user names, allows
 * the user to change his online status, and allows initiating
 * and receiving invitations to meetings.
 */
public class ExtLiveNamesApplet extends Applet
    implements LoginListener, ResolveListener,
               ItemListener, MeetingListener
{
```

```
private STSession m_session;
private CommunityService m_comm;
private AwarenessList m_awarenessList;
private Choice m_statusChoices;

private final String DISCONNECTED = "Disconnected";
private final String ACTIVE = "I Am Active";
private final String AWAY = "I Am Away";
private final String DND = "Do Not Disturb Me";

/**
 * Applet initialized. Create the session, load all components,
 * start the session and then login.
 */
public void init()
{
  try
  {
    m_session = new STSession("LiveNamesApplet " + this);
    m_session.loadAllComponents();
    m_session.start();

    setLayout(new BorderLayout());
    m_awarenessList = new AwarenessList(m_session, true);
    add(m_awarenessList, BorderLayout.CENTER);

    m_statusChoices = new Choice();
    m_statusChoices.setEnabled(false);
    m_statusChoices.addItem(DISCONNECTED);
    add(m_statusChoices, BorderLayout.SOUTH);
    m_statusChoices.addItemListener(this);

    ChatUI chatui = (ChatUI)m_session.getCompApi(ChatUI.COMP_NAME);
    chatui.addMeetingListener(this);

    AVController avController =
        new AVController(m_awarenessList.getModel());
    m_awarenessList.setController(avController);

    login();
```

```
    }
    catch(DuplicateObjectException e)
    {
      e.printStackTrace();
    }
  }

  /**
   * Login to the community using the user name and password
   * parameters from the html.
   */
  private void login()
  {
    m_comm = (CommunityService)
        m_session.getCompApi(CommunityService.COMP_NAME);
    m_comm.addLoginListener(this);
    m_comm.loginByPassword(getCodeBase().getHost(),
                           getParameter("loginName"),
                           getParameter("password"));
  }

  /**
   * Logged in event. Print logged in msg to console.
   * Resolve the users to add to the awareness list.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Logged In");

    m_awarenessList.addUser(
                    (STUser)event.getLogin().getMyUserInstance());

    m_statusChoices.setEnabled(true);
    m_statusChoices.removeAll();
    m_statusChoices.addItem(ACTIVE);
    m_statusChoices.addItem(AWAY);
    m_statusChoices.addItem(DND);

    LookupService lookup = (LookupService)
      m_session.getCompApi(LookupService.COMP_NAME);
```

```
  Resolver resolver =
      lookup.createResolver(false,  // Return all matches.
                            false,  // Non-exhaustive lookup.
                            true,   // Return resolved users.
                            false); // Do not return resolved groups.

  resolver.addResolveListener(this);

  String[] userNames = getUserNames();
  resolver.resolve(userNames);
}

/**
 * Helper method to read a list of user names from the html
 * parameter 'watchedNames'.
 */
String[] getUserNames()
{
  String users = getParameter("watchedNames");

  StringTokenizer tokenizer = new StringTokenizer(users, ",");
  String[] userNames = new String[tokenizer.countTokens()];

  int i = 0;
  while(tokenizer.hasMoreTokens())
  {
    userNames[i++] = tokenizer.nextToken();
  }
  return userNames;
}

/**
 * Users resolved succesfuly event. An event will be generated for
 * each resolved user. Add the resolved user to the awareness list.
 */
public void resolved(ResolveEvent event)
{
  m_awarenessList.addUser((STUser) event.getResolved());
}
```

```
/**
 * Handle a resolve conflict event. Will be received in the case
 * that more then one match was found for a specified user name.
 * Add the users to the awareness list anyway.
 */
public void resolveConflict(ResolveEvent event)
{
  STUser[] users = (STUser[]) event.getResolvedList();
  m_awarenessList.addUsers(users);
}

/**
 * Resolve failed. No users are available to add to the list.
 */
public void resolveFailed(ResolveEvent event)
{
}

/**
 * Called when the user chooses a status from the choice control
 */
public void itemStateChanged(ItemEvent event)
{
  if (event.getSource() == m_statusChoices)
  {
    STUserStatus status;

    if (event.getItem().equals(ACTIVE))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_ACTIVE,
                                0, ACTIVE);
    else if (event.getItem().equals(AWAY))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_AWAY,
                                0, AWAY);
    else if (event.getItem().equals(DND))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_DND,
                                0, DND);
    else return;

    if (m_comm.isLoggedIn())
```

```
      m_comm.getLogin().changeMyStatus(status);
    }
  }

  /**
   * Launch meeting event. Open a browser at the specified URL.
   */
  public void launchMeeting(MeetingInfo meetingInfo, URL url)
  {
    AppletContext context = getAppletContext();
    context.showDocument(url, "_blank");
  }

  /**
   * Meeting creation failed event.
   */
  public void meetingCreationFailed(MeetingInfo meetingInfo,
                                    int reason)
  {
    System.err.println("Create meeting failed reason = " + reason);
  }

  /**
   * Logged out event. Print logged out msg to console. Leave default
   * behavior which will display a dialog box.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");

    m_statusChoices.setEnabled(false);
    m_statusChoices.removeAll();
    m_statusChoices.addItem(DISCONNECTED);
  }

  /**
   * Applet destroyed. Logout, stop and unload the session.
   */
  public void destroy()
  {
```

```
    m_comm.logout();
    m_session.stop();
    m_session.unloadSession();
  }
}
```

User Status

To enable the user to change his online status, a UI element is needed to allow the user to choose between the three default online statuses: Active, Away, and Do Not Disturb (DND). A fourth status, Disconnected, indicates when the user is offline. Define four constants describing these statuses in the sample:

```
private final String DISCONNECTED = "Disconnected";
private final String ACTIVE = "I Am Active";
private final String AWAY = "I Am Away";
private final String DND = "Do Not Disturb Me";
```

Adding an AWT Choice Component

Add an AWT Choice component that provides the combo box the user can choose from. By default, disable the combo box and put it in the Disconnected state since that is the state the user is in before the applet logs in. In the sample's init() method these lines are added:

```
m_statusChoices = new Choice();
m_statusChoices.setEnabled(false);
m_statusChoices.addItem(DISCONNECTED);
add(m_statusChoices, BorderLayout.SOUTH);
m_statusChoices.addItemListener(this);
```

To the loggedIn() event handler, the following code was added to enable the Choice control, remove the disconnected option, and add the online status options:

```
m_statusChoices.setEnabled(true);
m_statusChoices.removeAll();
m_statusChoices.addItem(ACTIVE);
m_statusChoices.addItem(AWAY);
m_statusChoices.addItem(DND);
```

To the LoggedOut() event handler the code that returns the Choice control to the original Disconnected state is added:

```
m_statusChoices.setEnabled(false);
m_statusChoices.removeAll();
m_statusChoices.addItem(DISCONNECTED);
```

Changing the User Status

At this point no Sametime code has been included to change the user status. This code should be called whenever the user has selected a new status in the Choice control. The AWT defines an ItemListener interface for receiving such user selections, so the applet has been made an implementer of this interface and the applet is added as a listener to the `Choice` control (see the preceding subsection).

The ItemListener interface has only one event handler, itemStateChanged().
The following is the code for this event handler:

```
public void itemStateChanged(ItemEvent event)
{
  if (event.getSource() == m_statusChoices)
  {
    STUserStatus status;

    if (event.getItem().equals(ACTIVE))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_ACTIVE,
                                0, ACTIVE);
    else if (event.getItem().equals(AWAY))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_AWAY,
                                0, AWAY);
    else if (event.getItem().equals(DND))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_DND,
                                0, DND);
    else return;

    if (m_comm.isLoggedIn())
      m_comm.getLogin().changeMyStatus(status);
  }
}
```

This event handler code first verifies (for good programming practice) that the itemStateChanged() event came from the Choice control. If so, the code checks which status the user has selected and creates a new STUserStatus object to reflect this status. (STUserStatus is one of the core status types of the toolkit and is defined in the com.lotus.sametime.core.types package.) The different statuses have predefined constants.

At this point, you can change the user's online status by using a Login object representing the current login to Sametime. Once retrieved from the Community Services using the getLogin() method, you can use the Login object to change the online status, online attributes[4], and privacy settings of the user. In this sample code only the online status is changed. After ensuring you are still logged in, the code retrieves the Login object and calls its changeMyStatus() method to change the status to the just-created STUserStatus.

Changing User Status and Multiple Login

As mentioned in the Sametime User Model section of the previous chapter, a user can have multiple logins to Sametime by different clients. It is possible for one client to set the user status to Active and for a second client to set it to DND (Do Not Disturb). In Sametime, some user properties, such as online status and privacy, can have only a single value at a single point in time over the entire community.

The last client to set each of these properties sets it for *all* the user's clients.

All clients need to know when a user has changed, for example, his online status so that the clients can synchronize their UI and behavior with the latest status. This sample does not have code to handle this event. To observe this behavior, run a Sametime Connect client on the same machine the applet is running and log in as the same user you logged in from the applet. Change your status in the applet and note how the new status is automatically reflected at the bottom of the Sametime Connect client. Try doing the same from the Sametime Connect client; the applet does not exhibit this behavior. Instead, the applet displays a misleading status to the user.

---

[4] Online attributes are attributes we can set on ourselves and other users in the community can request to be subscribed to. For example, in Sametime 3.1 each Sametime Connect client uses online attributes to let other users know which audio and video tools they have.

To add this behavior to the sample, implement the MyStatusListener interface in the applet and add this listener to your Login object using the addMyStatusListener() method. These tasks are not included in the sample code. Adding this code is left to the reader as an exercise,.

Accepting Invitations and Launching the MRC

The second feature to add to this sample is the ability to initiate and join all types of meetings with your online coworkers.

The ChatUI component in the toolkit is the component responsible both for initiating and joining MRC meetings. This component defines the MeetingListener interface used specifically for these tasks. Use the following code to register the applet as a listener on these events in its init() method:

```
ChatUI chatui = (ChatUI)m_session.getCompApi(ChatUI.COMP_NAME);
chatui.addMeetingListener(this);
```

MeetingListener defines two event handlers that you need to implement in the applet. The first event handler, launchMeeting(), is called each time ChatUI needs to launch a new MRC meeting (whether initiated by you or from an invitation). Add this launch browser code to enable this event:

```
public void launchMeeting(MeetingInfo meetingInfo, URL url)
{
  AppletContext context = getAppletContext();
  context.showDocument(url, "_blank");
}
```

ChatUI passes the needed URL to open as a parameter, and the entire meeting details are available in the MeetingInfo parameter if needed. Launching a browser window and navigating it to a specific URL is simple for an applet, and two statements are sufficient to accomplish this task.

Once you have registered a MeetingListener to the ChatUI component, you are ready to receive MRC invitations. The ChatUI component displays a dialog allowing you to decide to accept or reject the invitation once it is sent to you. Adding the above code is all that is needed to accept MRC invitations.

The second event handler defined by MeetingListener is related only to the case when you initiate the meeting. This event handler is discussed in the next section.

Replacing the Awareness List Controller

Now that the code to launch the MRC has been added, other people can invite you to audio and video meetings. You can also initiate these meetings and invite other users.

If you right-click on the Awareness List while one or more online users are selected, a menu will display that gives you the option to chat with the selected users. You now need to add the menu items to start Audio, Video, Share, and Collaboration meetings and to check the users' available audio and video tools.

An object called a *controller* is associated with the awareness list. The controller defines the behavior of the list. The list consults the controller when it needs to know what to do if, for example, the user right-clicks on it. The default controller tells the list to open a popup menu that contains the Chat option only.

Creating an AVController Object

You can write your own controller, and change the behavior of the list completely. In this case, it is not necessary because the Sametime Toolkit provides a controller that supports the behavior needed. You have to tell the list that you want to use this controller instead of the standard one, in the init() method of the applet:

```
AVController avController =
    new AVController(m_awarenessList.getModel());
m_awarenessList.setController(avController);
```

This code simply creates a new controller object of type AVController and tells the list to use this controller instead of the default controller. Now, right clicking on the list provides all the new options we wanted, allowing the user to initiate these powerful meetings with online coworkers. Try it on this sample and see how easy it is to get all of this Sametime functionality inside your applet.

Detecting Meeting Creation Failure

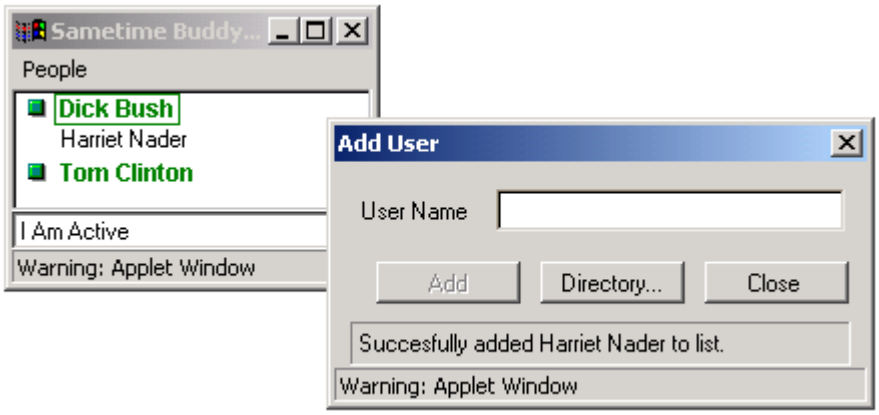
In some cases, meetings you initiate might fail to be created. To detect these cases, implement the meetingCreationFailed() event handler of MeetingListener. In the sample, we chose to print an error to the error stream.

```
public void meetingCreationFailed(MeetingInfo meetingInfo,
                                  int reason)
{
  System.err.println("Create meeting failed reason = " + reason);
}
```

Chapter 7. Buddy List Sample

Overview

In this chapter you will create a simple Buddy List sample as shown in the screens below.



To create a contact list, you need to add some important features and change some of the existing Extended Live Names applet behavior to address the following problems:

1. **You cannot dynamically add users because the list of users is currently hard coded into HTML as a parameter to the applet.**

   To solve this problem, use the Add dialog provided by the toolkit so you can add users simply by typing their names. The dialog resolves the names and displays any resolve conflicts. Because you no longer need the code that reads the users' list parameter from HTML and resolves the user names, you will delete that code in this chapter.

2. **If the list of users becomes dynamic, you need a way to store it until the next time we run the applet.**

   Use the Storage Service to store the contact list as an attribute on the Sametime server. The applet can remain unsigned (no need to access local disk storage), and the user will get his contact list from any machine or location.

3. **You need a way to control the privacy list in the community.**

   Adding privacy control is easy with the Privacy dialog provided by the toolkit. Instructions for this task are presented later in this chapter.

4. **The user should probably be prompted for his login name and password.**

   In the Extended Live Names applet, the login name and password are passed as HTML parameters to the applet. A real contact list application will probably require the user to enter this information in a login dialog. Since creating this dialog requires straightforward Java programming and is not specific to Sametime, this tutorial will not demonstrate creation of a login dialog.

Fortunately, the Sametime Java Toolkit provides UI components that make adding these features easy.

For instructions on how to run this sample, see Chapter 3.

Buddy List HTML Code

The following is the HTML code for the Buddy List Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime BuddyList Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=BuddyListApplet.class NAME=BuddyListApplet
            WIDTH=95% HEIGHT=95%>
<PARAM NAME='archive' VALUE='../CommRes.jar,../STComm.jar'>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
</APPLET>
</BODY>
</HTML>
```

Applet Source Code

The following is the source code for the Buddy List Sample:

BuddyListApplet.java

```
import java.awt.*;
import java.applet.*;
import java.util.*;
import java.net.URL;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.community.*;
import com.lotus.sametime.chatui.ChatUI;
import com.lotus.sametime.chatui.MeetingListener;
import com.lotus.sametime.chatui.MeetingInfo;

/**
 * Sample applet that displays a simple buddy list in a popup frame.
 */
public class BuddyListApplet extends Applet
    implements LoginListener, MeetingListener
{
```

```
private STSession m_session;
private CommunityService m_comm;
private BuddyListFrame m_frame;

/**
 * Applet initialized. Create the session, load all components,
 * start the session and then login.
 */
public void init()
{
  try
  {
    m_session = new STSession("BuddyListApplet " + this);
    m_session.loadAllComponents();
    m_session.start();

    ChatUI chatui = (ChatUI)m_session.getCompApi(ChatUI.COMP_NAME);
    chatui.addMeetingListener(this);

    m_frame = new BuddyListFrame(m_session);
    login();
  }
  catch(DuplicateObjectException e)
  {
    e.printStackTrace();
  }
}

/**
 * Login to the community using the user name and password
 * parameters from the html.
 */
private void login()
{
  m_comm = (CommunityService)
      m_session.getCompApi(CommunityService.COMP_NAME);
  m_comm.addLoginListener(this);

  m_comm.loginByPassword(getCodeBase().getHost(),
                       getParameter("loginName"),
```

```
                        getParameter("password"));

  }

  /**
   * Logged in event. Create the buddy list frame and show it.
   */
  public void loggedIn(LoginEvent evt)
  {
    m_frame.setVisible(true);
  }

  /**
   * Launch meeting event. Open a browser at the specified URL.
   */
  public void launchMeeting(MeetingInfo meetingInfo, URL url)
  {
    AppletContext context = getAppletContext();
    context.showDocument(url, "_blank");
  }

  /**
   * Meeting creation failed event.
   */
  public void meetingCreationFailed(MeetingInfo meetingInfo,
                                    int reason)
  {
    System.err.println("Create meeting failed reason = " + reason);
  }

  /**
   * Logged out event. Hide the buddy list frame. Leave default
   * behavior which will display a dialog box.
   */
  public void loggedOut(LoginEvent event)
  {
    if (m_frame != null)
      m_frame.setVisible(false);
  }
```

```
  /**
   * Applet destroyed. Logout, stop and unload the session.
   */
  public void destroy()
  {
    m_comm.logout();
    m_session.stop();
    m_session.unloadSession();

    m_frame.dispose();
  }
}
```

BuddyListFrame.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;
import java.util.*;
import java.net.URL;

import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.core.comparch.DuplicateObjectException;
import com.lotus.sametime.core.constants.STError;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.storage.*;
import com.lotus.sametime.awarenessui.list.AwarenessList;
import com.lotus.sametime.awarenessui.av.AVController;
import com.lotus.sametime.commui.*;

/**
 * The buddy list frame. Part of Buddy List Applet sample.
 */
public class BuddyListFrame extends Frame
    implements LoginListener, ResolveViewListener,
               ActionListener, ItemListener,
               StorageServiceListener

{
```

```
private STSession m_session;
private CommunityService m_commService;
private StorageService m_storageService;
private Integer m_nReqID;
private AwarenessList m_awarenessList;
private MenuItem m_menuAddToList;
private MenuItem m_menuRemoveFromList;
private MenuItem m_menuWhoCanSeeMe;
private Choice m_statusChoices;

private final static int BUDDY_LIST_ATT_KEY = 0xFFFF;

private final String DISCONNECTED = "Disconnected";
private final String ACTIVE = "I Am Active";
private final String AWAY = "I Am Away";
private final String DND = "Do Not Disturb Me";

/**
 * BuddyListFrame constructor
 */
public BuddyListFrame(STSession session)
{
  super("Sametime Buddy List");
  m_session = session;

  m_commService = (CommunityService)
    m_session.getCompApi(CommunityService.COMP_NAME);
  m_commService.addLoginListener(this);

  m_storageService = (StorageService)
    m_session.getCompApi(StorageService.COMP_NAME);
  m_storageService.addStorageServiceListener(this);

  addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e)
      {
        storeBuddyList();
        dispose();
      }
    });
```

```
    init();
  }

  /**
   * Setup the buddy list frame.
   */
  public void init()
  {
    setLayout(new BorderLayout());

    m_awarenessList = new AwarenessList(m_session, true);
    add(m_awarenessList, BorderLayout.CENTER);

    m_statusChoices = new Choice();
    m_statusChoices.setEnabled(false);
    m_statusChoices.addItem(DISCONNECTED);
    add(m_statusChoices, BorderLayout.SOUTH);
    m_statusChoices.addItemListener(this);

    AVController avController =
        new AVController(m_awarenessList.getModel());
    avController.enableDelete(true);
    m_awarenessList.setController(avController);

    m_menuAddToList = new MenuItem("Add to List...");
    m_menuAddToList.addActionListener(this);

    m_menuRemoveFromList = new MenuItem("Remove from List");
    m_menuRemoveFromList.addActionListener(this);

    m_menuWhoCanSeeMe = new MenuItem("Who Can See Me...");
    m_menuWhoCanSeeMe.addActionListener(this);

    Menu menuOptions = new Menu("People");
    menuOptions.add(m_menuAddToList);
    menuOptions.add(m_menuRemoveFromList);
    menuOptions.add(m_menuWhoCanSeeMe);

    MenuBar menuBar = new MenuBar();
```

```
    menuBar.add(menuOptions);
    setMenuBar(menuBar);

    pack();

    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    setLocation((int)((d.width - getSize().width)/2),
                (int)((d.height - getSize().height)/2));
  }
  /**
   * Logged in event. Update the choice control with all the
   * status options.
   */
  public void loggedIn(LoginEvent event)
  {
    System.out.println("Logged In");
     m_awarenessList.addUser(
            (STUser)event.getLogin().getMyUserInstance());

    m_statusChoices.setEnabled(true);
    m_statusChoices.removeAll();
    m_statusChoices.addItem(ACTIVE);
    m_statusChoices.addItem(AWAY);
    m_statusChoices.addItem(DND);
  }

  /**
   * Logged out event. Reset the choice control to show
   * "Disconnected" and disabled.
   */
  public void loggedOut(LoginEvent event)
  {
    System.out.println("Logged Out");

    m_statusChoices.setEnabled(false);
    m_statusChoices.removeAll();
    m_statusChoices.addItem(DISCONNECTED);
  }

  /**
```

```
 * Called when the user chooses a status from the choice control
 */
public void itemStateChanged(ItemEvent event)
{
  if (event.getSource() == m_statusChoices)
  {
    STUserStatus status;

    if (event.getItem().equals(ACTIVE))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_ACTIVE,
                                0, ACTIVE);
    else if (event.getItem().equals(AWAY))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_AWAY,
                                0, AWAY);
    else if (event.getItem().equals(DND))
      status = new STUserStatus(STUserStatus.ST_USER_STATUS_DND,
                                0, DND);
    else return;

    if (m_commService.isLoggedIn())
      m_commService.getLogin().changeMyStatus(status);
  }
}

/**
 * Load the buddy list from the server using the Storage Service.
 */
void loadBuddyList()
{
  m_nReqID = m_storageService.queryAttr(BUDDY_LIST_ATT_KEY);
}

/**
 * Load the buddy list from an attribute returned by the Storage
 * Service.
 */
void loadBuddyListFromAttribute(STAttribute attr)
{
  STUser user;
  String name;
```

```
  String id;

  StringTokenizer tokenizer =
    new StringTokenizer(attr.getString(),";");

  while(tokenizer.countTokens() >= 2)
  {
    name = tokenizer.nextToken();
    id = tokenizer.nextToken();

    user = new STUser(new STId(id, ""), name, "");
    m_awarenessList.addUser(user);
  }
}

/**
 * Store the buddy list to the server using the Storage Service.
 */
void storeBuddyList()
{
  STUser[] usersList = m_awarenessList.getItems();

  StringBuffer buffer = new StringBuffer();
  for(int i=0; i<usersList.length; i++)
  {
    buffer.append(usersList[i].getName());
    buffer.append(";");
    buffer.append(usersList[i].getId().getId());
    buffer.append(";");
  }

  STAttribute attribute = new STAttribute(BUDDY_LIST_ATT_KEY,
                                          buffer.toString());

  m_nReqID = m_storageService.storeAttr(attribute);
}

/**
 * Action event listener. Called when one of the menus is selected.
 */
```

```
public void actionPerformed(ActionEvent event)
{
  Object src = event.getSource();

  if (src == m_menuAddToList)
    addToList();
    else if(src == m_menuRemoveFromList)
     removeFromList();
  else if (src == m_menuWhoCanSeeMe)
    whoCanSeeMe();
}

/**
 * Show the add dialog.
 */
void addToList()
{
  AddDialog addDialog = new AddDialog(this, m_session, "Add User");
  addDialog.addResolveViewListener(this);
  addDialog.setVisible(true);
}

void removeFromList()
{
   m_awarenessList.removeUsers(m_awarenessList.getSelectedItems());
}
/**
 * Show the privacy (who-can-see-me) dialog.
 */
void whoCanSeeMe()
{
  PrivacyDialog dialog = new PrivacyDialog(this, m_session);
  dialog.setVisible(true);
}

/**
 * A user resolve request from the add dialog was successful.
 */
public void resolved(ResolveViewEvent event)
{
```

```
    STUser user = event.getUser();
    m_awarenessList.addUser(user);
  }

  /**
   * A resolve request from the add dialog failed.
   */
  public void resolveFailed(ResolveViewEvent event)
  {
    System.out.println("Couldn't find user. Reason = " +
                                          event.getReason());
  }

  /**
   * Called as a response to a query attribute request.
   */
  public void attrQueried(StorageEvent event)
  {
    if (m_nReqID == event.getRequestId())
    {
      if(event.getRequestResult() == STError.ST_OK)
      {
        loadBuddyListFromAttribute((STAttribute)
                event.getAttrList().firstElement());
      }
      else
      {
        System.out.println("Couldn't load buddy list");
      }
    }
  }

  /**
   * A response from the servr to a store attribute request.
   */
  public void attrStored(StorageEvent event)
  {
    if (m_nReqID == event.getRequestId() &&
        event.getRequestResult() != STError.ST_OK)
    {
```

```
      System.out.println("Couldn't store buddy list");
    }
  }

  /**
   * Indicates the the Storage Service is now available
   */
  public void serviceAvailable(StorageEvent event)
  {
    loadBuddyList();
  }

  /**
   * Indicates that the Storage Service is unavailable.
   */
  public void serviceUnavailable(StorageEvent event)
  {
  }

  /**
   * Indicates that one or more storage attributes of this login
   * were modified by a different login.
   */
  public void attrUpdated(StorageEvent event)
  {
  }
}
```

The Buddy List Applet

The Buddy List applet code creates the session object as usual in its init() method and implements two listeners:

- LoginListener – Receives loggedIn() and loggedOut() events.
- MeetingListener – Handles the launchMeeting() event so the applet can support MRC meetings.

Creating and Destroying the Buddy List Frame

The applet creates the Buddy List frame in the init() method. In the loggedIn() event handler you create the Buddy List frame and make it visible. Note that the applet does not display any UI elements except for this frame.

In the loggedOut() event handler, you hide the Buddy List frame. The following code shows these two event handlers:

```
public void loggedIn(LoginEvent evt)
{
  m_frame.setVisible(true);
}

public void loggedOut(LoginEvent event)
{
  if (m_frame != null)
    m_frame.setVisible(false);
}
```

The destroy() method of the applet contains the command to dispose of the Buddy List frame:

```
m_frame.dispose();
```

The Buddy List Frame

The Buddy List Frame is the only UI element displayed by the Buddy List applet and contains most of the Buddy List's functionality.

Constructing the Buddy List Frame

The Buddy List frame constructor stores the Sametime session object and adds the frame as a LoginListener to the Community Service and as a StorageServiceListener to the Storage Service. It then declares an anonymous inline class and adds it as a WindowListener to receive the windowClosing() event. In this event handler, the frame constructor calls a method to store the frame's Buddy List and then disposes of the frame. Finally, the constructor calls the init() method to lay out all the AWT components of the frame.

The following is the frame constructor code:

```
public BuddyListFrame(STSession session)
{
  super("Sametime Buddy List");
  m_session = session;

  m_commService = (CommunityService)
    m_session.getCompApi(CommunityService.COMP_NAME);
  m_commService.addLoginListener(this);

  m_storageService = (StorageService)
    m_session.getCompApi(StorageService.COMP_NAME);
  m_storageService.addStorageServiceListener(this);

  addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e)
      {
        storeBuddyList();
        dispose();
      }
    });

  init();
}
```

Initializing the Buddy List Frame

The Buddy List Frame init() method does the following:

- Creates and adds the AwarenessList to the frame layout.
- Adds the Choice control to allow the user to select his status.
- Changes the default controller of the list to an AVController so the user can initiate MRC meetings. The AVController is also modified to enable removal of users from the awareness list.
- Creates and adds a menu bar with three menu options: one to open the Add dialog, one to remove users from the list, and one to open the Privacy dialog.
- Packs all the AWT components and moves the frame to the center of the screen.

The following is the init() method code:

```
public void init()
{
  setLayout(new BorderLayout());

  m_awarenessList = new AwarenessList(m_session, true);
  add(m_awarenessList, BorderLayout.CENTER);

  m_statusChoices = new Choice();
  m_statusChoices.setEnabled(false);
  m_statusChoices.addItem(DISCONNECTED);
  add(m_statusChoices, BorderLayout.SOUTH);
  m_statusChoices.addItemListener(this);

  AVController avController =
      new AVController(m_awarenessList.getModel());
  avController.enableDelete(true);
  m_awarenessList.setController(avController);

  m_menuAddToList = new MenuItem("Add to List...");
  m_menuAddToList.addActionListener(this);

  m_menuRemoveFromList = new MenuItem("Remove from List");
   m_menuRemoveFromList.addActionListener(this);

  m_menuWhoCanSeeMe = new MenuItem("Who Can See Me...");
  m_menuWhoCanSeeMe.addActionListener(this);

  Menu menuOptions = new Menu("People");
  menuOptions.add(m_menuAddToList);
   menuOptions.add(m_menuRemoveFromList);
  menuOptions.add(m_menuWhoCanSeeMe);

  MenuBar menuBar = new MenuBar();
  menuBar.add(menuOptions);

  setMenuBar(menuBar);

  pack();


  Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
  setLocation((int)((d.width - getSize().width)/2),
              (int)((d.height - getSize().height)/2));
```

```
}
```

Handling Status Changes

The loggedIn(), loggedOut(), and itemStateChanged() event handlers are taken from the previous chapter to handle the Choice control. Note that two different listeners implement LoginListener in the same applet; the toolkit supports this kind of coding requirement easily.

The Add Dialog

The code from the previous chapter that read the WatchedNames parameter and resolved the names in it was removed. Instead, the Add dialog will be used to add users to the contact list.

Catching Menu Events

The actionPerformed() event handler, which is called by the AWT when a menu item is clicked, checks which menu item originated the call. For example, if the user clicked the "Add to List"' menu item, the addToList() method is called to allow the user to add users to the list:

```
public void actionPerformed(ActionEvent event)
{
  Object src = event.getSource();

  if (src == m_menuAddToList)
    addToList();
    else if(src == m_menuRemoveFromList)
      removeFromList();
  else if (src == m_menuWhoCanSeeMe)
    whoCanSeeMe();
}
```

Creating the Add Dialog

The addToList() method displays the Add dialog. The Add dialog, which is part of the com.lotus.sametime.commui package, allows the user to type in a name and resolves that name for the user. If multiple matches are found, the user is prompted to choose between them. The following code creates the Add dialog, adds the frame as a listener to it, and displays it:

```
void addToList()
{
  AddDialog addDialog = new AddDialog(this, m_session, "Add User");
  addDialog.addResolveViewListener(this);
  addDialog.setVisible(true);
}
```

Handling ResolveViewListener Events

The Add dialog sends notifications through the two event handlers of ResolveViewListener: resolved() and resolveFailed().

The resolved() event handler is called to inform you that the specified user name was resolved successfully, or if multiple matches were found, that the user chose the name he meant from a list presented to him by the dialog. In this sample, the resolved user is added to the awareness list. The following is the resolved() event handler code:

```
public void resolved(ResolveViewEvent event)
{
  STUser user = event.getUser();
  m_awarenessList.addUser(user);
```

```
}
```

The resolveFailed() method is called if the name was not found or if some other problem occurred when trying to resolve this name. A full-scale application would probably display a message box to inform the user that something went wrong. In this sample, a message is printed to the Java Console:

```
public void resolveFailed(ResolveViewEvent event)
{
  System.out.println("Couldn't find user. Reason = " +
                                        event.getReason());
}
```

Loading and Storing the Contact List

Once the contact list has been created, it should be stored so if the user returns to the applet at a later time he will not need to create it again. The Storage Service was specifically designed to store such user-related information, or attributes, on the Sametime server.

The Storage Service allows you to store and query user-related information or attributes directly on the Sametime server. The attributes are stored per user; one user cannot access the attributes of another user. The service allows you to access this information from wherever you log in to your Sametime server. You need to define an *attribute key* that will identify the attribute you want to store. The *value* of the attribute can be any kind of data: Boolean, integer, string, or binary data. Every Sametime attribute has a key and a value.

Storing the Contact List

In a previous section the Buddy List frame constructor declared an anonymous inline class and added it as a WindowListener to receive the windowClosing() event. In this event handler the frame constructor calls the storeBuddyList() method to store the frame's contact list and then disposes of the frame.

The storeBuddyList() method does the following:

Retrieves the STUser objects of all the users in the awareness list.

Dumps the user names and user IDs of the users to one long string using the semicolon (;) character as a delimiter.

Turns the string into an STAttribute object.

Calls the Storage Service method storeAttr() to store the attribute.

The following is the code for the storeBuddyList()method:

```
void storeBuddyList()
{
  STUser[] usersList = m_awarenessList.getItems();

  StringBuffer buffer = new StringBuffer();
  for(int i=0; i<usersList.length; i++)
  {
    buffer.append(usersList[i].getName());
    buffer.append(";");
    buffer.append(usersList[i].getId().getId());
    buffer.append(";");
  }

  STAttribute attribute = new STAttribute(BUDDY_LIST_ATT_KEY,
                                          buffer.toString());
```

```
    m_nReqID = m_storageService.storeAttr(attribute);
  }
```

The storeAtrr() method returns the store request ID that is stored in the member variable m_nReqID. As a response to the storeAttr() request, an attrStored() event will always be generated, whether the original request succeeds or fails.

```
  public void attrStored(StorageEvent event)
  {
    if (m_nReqID == event.getRequestId() &&
        event.getRequestResult() != STError.ST_OK)
    {
      System.out.println("Couldn't store buddy list");
    }
  }
```

In the above code, a check is made to ensure that the generated event is a response to the store request by comparing the event's Request ID with the original Request ID. If so, and the store request was successful, nothing is done. If the request failed for some reason, a message is printed to the Java Console.

Loading the Contact List

You need to query the Storage Service to retrieve stored attributes. In our sample, you need to load the contact list right after the frame is created so the user will have the contact list he used the last time he ran the applet. You also need to ensure that the Storage Service is available before attempting to load the contact list.

The Storage Service has two events that provide information about the current status of the service. The serviceAvailable() event handler is called to indicate that the Storage Service is ready to receive store and query requests. The serviceUnavailable() event handler is called to indicate that the Storage Service is currently not ready to receive and reply to requests. Use the serviceAvailable() event to trigger the call to the loadBuddyList() method when you know that the Storage Service is ready to receive requests:

```
  public void serviceAvailable(StorageEvent event)
  {
    loadBuddyList();
  }

  public void serviceUnavailable(StorageEvent event)
  {
  }
```

The loadBuddyList() method simply queries the Storage Service for the attribute with the key BUDDY_LIST_ATT_KEY defined previously and used when storing the contact list:

```
  void loadBuddyList()
  {
    m_nReqID = m_storageService.queryAttr(BUDDY_LIST_ATT_KEY);
  }
```

The query request ID is recorded so you can compare it with the attrQueried() response. You will get this response for both a successful and failed query request:

```
  public void attrQueried(StorageEvent event)
  {
    if (m_nReqID == event.getRequestId())
    {
      if(event.getRequestResult() == STError.ST_OK)
      {
        loadBuddyListFromAttribute((STAttribute)
                event.getAttrList().firstElement());
      }
      else
```

```
      {
        System.out.println("Couldn't load buddy list");
      }
    }
  }
```

In our sample, you first check to ensure that the generated event is a response to your query request. If so, and if the query was not successful, you print an error message to the Java Console. If the query was successful, call the loadBuddyListFromAttribute() method with the first attribute in the attribute list held by the event object. Since the query was for a single attribute key, you know that you will always receive one attribute in the response. Here is the loadBuddyListFromAttribute() method code:

```
  void loadBuddyListFromAttribute(STAttribute attr)
  {
    STUser user;
    String name;
    String id;

    StringTokenizer tokenizer =
      new StringTokenizer(attr.getString(),";");

    while(tokenizer.countTokens() >= 2)
    {
      name = tokenizer.nextToken();
      id = tokenizer.nextToken();

      user = new STUser(new STId(id, ""), name, "");
      m_awarenessList.addUser(user);
    }
  }
```

In this method, the returned attribute is parsed using a standard tokenizer, and the users in it are added to the awareness list.

Receiving Attribute Update Events

The attrUpdated() event is also generated by StorageServiceListener. This event handler is called if the user has multiple clients logged in at the same time to a community, and one of the clients stores an attribute using the Storage Service. This event is generated in all the other clients to inform them that another client changed an attribute value. The application can then decide, according to the attribute key of the changed attribute (the changed attribute value is not passed in this event), whether to query for the attribute value and do any further steps with the returned value. In this sample this event is ignored:

```
  public void attrUpdated(StorageEvent event)
  {
  }
```

Privacy Dialog

In the actionPerformed() event handler we check to see if the “Who Can See Me” menu item was selected and, if so, call the whoCanSeeMe() method:

```
  else if (src == m_menuWhoCanSeeMe)
    whoCanSeeMe();
```

The whoCanSeeMe() method creates a new Privacy dialog and displays it to the user. From that point on, the Privacy dialog handles any user selections and sets the user’s privacy accordingly. The Privacy dialog does not require any listener.

```
  void whoCanSeeMe()
```

```
{
  PrivacyDialog dialog = new PrivacyDialog(this, m_session);
  dialog.setVisible(true);
}
```

If you choose to not allow a user to know you are online using the Privacy dialog, you will also *not* be able to see his online status. Sametime privacy is symmetric: if you do not let someone see your status, you will not be able to see his status.

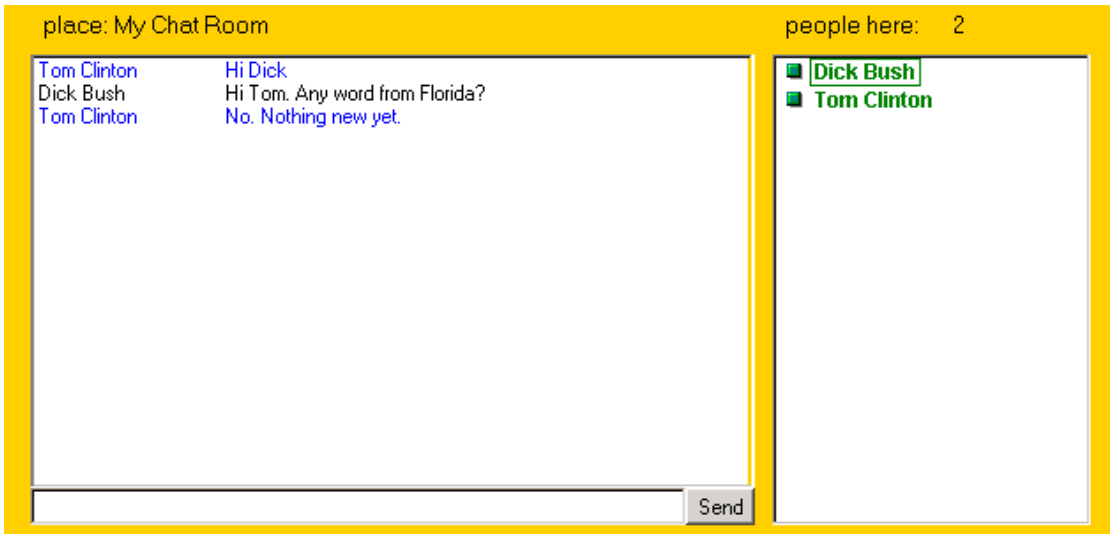
Part III

Part III describes more advanced features of the Sametime Java Toolkit, such as the Places Service, a core service of Sametime architecture. This part of the tutorial will show how to use the Places Service to build a meeting applet in which Sametime users can meet and collaborate.

Chapter 8. Chat Meeting Sample

Overview

In this chapter you will build a chat-only meeting applet. This applet will allow users to meet in a predefined virtual place, be aware of each other (that is, see who else is in the place), and use text to chat with each other.



This chapter also describes the Places Service, which is the foundation of the sample. You will see how you can enter a place and track other users entering and leaving the place. You will also learn how to use some of the UI utility classes provided by the toolkit to speed up writing of Sametime-enabled applications.

For instructions on how to run this sample, see the section "_____" in Chapter 3.

Chat Meeting HTML Code

The following is the HTML code for the Chat Meeting Sample:

```
<HTML>
<HEAD>
<TITLE>Sample Sametime Chat Meeting Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE=ChatMeeting.class NAME=ChatMeeting WIDTH=95% HEIGHT=95%>
<PARAM NAME='archives' VALUE='../CommRes.jar,../STComm.jar>
<PARAM NAME='loginName' VALUE='Tom'> <!-- REPLACE -->
<PARAM NAME='password' VALUE='sametime'> <!-- REPLACE -->
<PARAM NAME='placeName' VALUE='My Meeting Room'>
</APPLET>
</BODY>
```

```
</HTML>
```

Chat Meeting Source Code

The following is the source code for the Chat Meeting Sample:

ChatMeeting.java

```
import java.awt.*;
import java.applet.*;
import java.awt.image.*;
import java.awt.event.*;

import com.lotus.sametime.core.comparch.*;
import com.lotus.sametime.core.constants.*;
import com.lotus.sametime.core.types.*;
import com.lotus.sametime.community.*;
import com.lotus.sametime.places.*;
import com.lotus.sametime.awarenessui.*;
import com.lotus.sametime.awarenessui.placelist.*;
import com.lotus.sametime.util.*;

/**
 * Meeting Applet sample showing how to create a chat only meeting
 */
public class ChatMeeting extends Applet
{
  private STSession m_session;
  private CommunityService m_comm;
  private Place m_place;
  private PlaceAwarenessList m_peopleList;

  private ChatPanel m_chatPanel;
  private Label m_PeopleNumLbl;

  private static int numUsersInPlace = 0;

  /**
   * Initialize the applet. Create the session, load and start
   * the components.
   */
```

```
public void init()
{
  try
  {
    m_session = new STSession("ChatMeeting " + this);
  }
  catch (DuplicateObjectException exception)
  {
    exception.printStackTrace();
  }

  m_session.loadAllComponents();
  m_session.start();

  m_comm = (CommunityService)m_session.getCompApi
               (CommunityService.COMP_NAME);
  m_comm.addLoginListener(new CommunityEventsListener());

  String community = getCodeBase().getHost().toString();
  String loginName = getParameter("loginName");
  String password = getParameter("password");
  m_comm.loginByPassword(community, loginName, password);
}
/**
 * Enter the place the meeting will take place in.
 */
public void enterPlace()
{
  PlacesService placesService =
    (PlacesService)m_session.getCompApi(PlacesService.COMP_NAME);

  m_place = placesService.createPlace(
                getParameter("placeName"), // place unique name
                getParameter("placeName"), // place display name
                EncLevel.ENC_LEVEL_DONT_CARE, // encryption level
                0,                            // place type
                PlacesConstants.PLACE_PUBLISH_DONT_CARE);

  m_place.addPlaceListener(new PlaceEventsListener());
  m_place.enter();
```

```
}

/**
 * Layout the applet UI.
 */
protected void layoutAppletUI()
{
  removeAll();
  this.setBackground(new Color(0xFFcc00));
  setLayout(new BorderLayout(10,0));

  Panel eastPnl = new Panel();
  eastPnl.setLayout(new BorderLayout());

  Panel NPanel = new Panel(new BorderLayout());
  Panel peopleHerePnl = new Panel();
  peopleHerePnl.setLayout(new FlowLayout(FlowLayout.LEFT));

  Label PeopleHereLbl = new Label("People Here:");
  PeopleHereLbl.setFont(new Font("Dialog",Font.PLAIN,14));
  peopleHerePnl.add(PeopleHereLbl);

  m_PeopleNumLbl = new Label("0");
  m_PeopleNumLbl.setFont(new Font("Dialog",Font.PLAIN,14));
  peopleHerePnl.add(m_PeopleNumLbl);

  NPanel.add(peopleHerePnl, BorderLayout.NORTH);

  m_peopleList = new PlaceAwarenessList(m_session, true);
  m_peopleList.addAwarenessViewListener(
                    new ParticipantListListener());

  eastPnl.add(NPanel,BorderLayout.NORTH);
  eastPnl.add(m_peopleList,BorderLayout.CENTER);

  Panel chatPnl = new Panel();
  chatPnl.setLayout(new BorderLayout());

  Panel chatLblPnl = new Panel(new FlowLayout(FlowLayout.LEFT));
  Label chatLbl = new Label("Place: " +
```

```
                          getParameter("placeName"), Label.LEFT);
    chatLbl.setFont(new Font("Dialog",Font.PLAIN,14));
    chatLblPnl.add(chatLbl);
    chatPnl.add(chatLblPnl,BorderLayout.NORTH);

    m_chatPanel = new ChatPanel(m_session, getAppletContext());
    chatPnl.add(m_chatPanel,BorderLayout.CENTER);

    add(chatPnl,BorderLayout.CENTER);
    add(eastPnl, BorderLayout.EAST);

    validate();
  }

  /**
   * A listener for loggedIn/loggedOut events.
   */
  class CommunityEventsListener implements LoginListener
  {
    public void loggedIn(LoginEvent event)
    {
      layoutAppletUI();
      enterPlace();
    }

    public void loggedOut(LoginEvent event)
    {
    }
  }

  /**
   * A listener for place events.
   */
  class PlaceEventsListener extends PlaceAdapter
  {
    public void entered(PlaceEvent event)
    {
      m_peopleList.bindToSection(m_place.getMySection());
      m_chatPanel.bindToPlace(event.getPlace());
    }
```

```
}

/**
 * A listener for participant list events.
 */
class ParticipantListListener extends AwarenessViewAdapter
{
  boolean firstTime = true;

  public void usersAdded(AwarenessViewEvent event)
  {
    if (firstTime)
    {
      firstTime = false;
      numUsersInPlace = event.getUsers().length;
    }
    else numUsersInPlace++;

    setLabel(numUsersInPlace);
  }

  public void usersRemoved(AwarenessViewEvent event)
  {
    numUsersInPlace--;
    setLabel(numUsersInPlace);
  }

  private void setLabel(int numOfPeople)
  {
    m_PeopleNumLbl.setText(String.valueOf(numOfPeople));
    m_PeopleNumLbl.validate();
  }
}

/**
 * Applet destroyed. Logout, stop and unload the session.
 */
public void destroy()
{
  m_comm.logout();
```

```
    m_session.stop();
    m_session.unloadSession();
  }
}
```

ChatPanel.java

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.applet.*;

import com.lotus.sametime.core.types.*;
import com.lotus.sametime.core.comparch.STSession;
import com.lotus.sametime.community.*;
import com.lotus.sametime.places.*;
import com.lotus.sametime.guiutils.chat.*;

public class ChatPanel extends Panel implements ActionListener
{
  private STSession m_session;
  private Place m_place;

  private AppletContext m_context;

  private ChatArea m_chatTranscript;
  private TextField m_txtField;
  private Font m_font;
  private Button m_btnSend;
  private Panel m_sendPanel;

  /**
   * Chat panel constructor.
   */
  public ChatPanel(STSession session, AppletContext context)
  {
    m_session = session;
    m_context = context;
    init();
  }
```

```
/**
 * Bind the chat panel to a specific place.
 */
public void bindToPlace(Place place)
{
  m_place = place;
  MyselfInPlace me = m_place.getMyselfInPlace();
  me.addMyMsgListener(new MyMsgEventsListener());
}
/**
 * Layout the chat panel UI.
 */
private void init()
{
  setLayout(new BorderLayout());

  m_font = new Font("Dialog",Font.PLAIN ,12);
  m_chatTranscript = new ChatArea(
                        1000,  //chatarea buffer capacity
                        m_font,
                        17);   //max chat trans name width
  m_chatTranscript.addChatAreaListener(
                     new ChatAreaEventsListener());

  m_sendPanel = new Panel();
  m_sendPanel.setLayout(new BorderLayout());

  m_btnSend = new Button("Send");
  m_btnSend.addActionListener (this);
  m_sendPanel.add("East", m_btnSend);

  m_txtField = new TextField();
  m_txtField.addKeyListener(new TextFieldEventsListener());
  m_txtField.setBackground(Color.white);

  m_sendPanel.add("Center", m_txtField);

  add("Center", m_chatTranscript);
  add("South", m_sendPanel);
}
```

```
/**
 * Send the text in the msg parameter to the section.
 */
public void sendText(String msg)
{
  if (m_place != null)
  {
    Section mySection = m_place.getMySection();
    mySection.sendText(msg);
    m_txtField.requestFocus();
  }
}

/**
 * Send button pressed.
 */
public void actionPerformed(ActionEvent event)
{
  if (event.getSource() == m_btnSend)
  {
    String msg  = m_txtField.getText();
    m_txtField.setText("");
    sendText(msg);
  }
}

/**
 * Respond to text received event and display it.
 */
class MyMsgEventsListener extends MyMsgAdapter
{
  public void textReceived(MyselfEvent event)
  {
    PlaceMember sender = event.getSender();
    String text = event.getText();

    if (!(sender instanceof UserInPlace))
      return;
```

```
    UserInPlace userSender = (UserInPlace)sender;
    CommunityService commService =
      (CommunityService)m_session.
              getCompApi(CommunityService.COMP_NAME);
    boolean myText = (commService.getLogin().
      getMyUserInstance().getLoginId().
                  equals(userSender.getLoginId()));
    String userName = userSender.getName();

    m_chatTranscript.write(userName, text,
            (myText? Color.blue: Color.black ));
  }
}

/**
 * Respond to the URL clicked event.
 */
class ChatAreaEventsListener extends ChatAreaAdapter
{
  public void chatURLClicked(ChatAreaEvent event)
  {
    try {
      String url = event.getURL();
      if (url.indexOf("//") == -1)
        url = "http://" + url;
      m_context.showDocument(new URL(url), "_blank");
    }
    catch(MalformedURLException mue)
    {
      mue.printStackTrace();
    }
         }
}
/**
 * Receive the text submit event and pass it on.
 */
class TextFieldEventsListener extends KeyAdapter
{
  public void keyPressed(KeyEvent key)
  {
```

```
      if (key.getKeyCode() == key.VK_ENTER)
      {
        sendText(m_txtField.getText());
        m_txtField.setText("");
      }
    }
  }
}
```

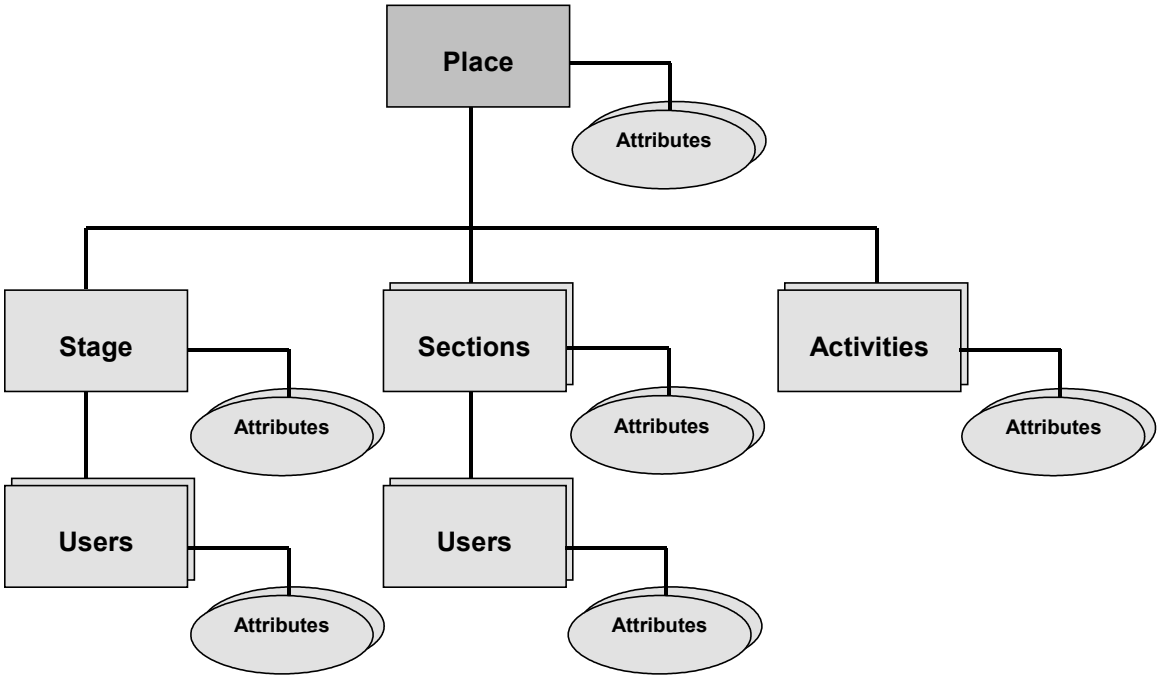
The Places Architecture

The Places Service exposes the places architecture and allows you to create places and add meeting activities to them. The places architecture defines the concept of a virtual place where users can meet and collaborate. A place consists of sections, activities, and attributes.

Place

A *place* is uniquely identified by its *name* and *type*. To the Sametime server, the name of a particular place is just a string. The application developer chooses an appropriate name when he creates the place in his application.

Applications that allow the user to be in more than one place must choose the namespace of all the places the user could possibly be in. For example, one application could make URLs the place names, and another could make the Lotus Notes® document ID the place name. By choosing the correct identifier for a place, you define the scope of the place in your application. For example, you can define whether people must be looking at the same document or be in the same database in order to see and collaborate with other users using the same application.



Sections

A place consists of a number of *sections*. The first section is a special section called the *stage*. Users on the stage have special permissions, such as the ability to "speak" to the entire place and not just to their own section. Users need to enter the place to take advantage of the services provided by the Places Service. Once a user successfully enters a place, he is always inside one of the place sections. A user can register to receive the list of all available sections in a place and with the list he or she can move from one section to another.

Activities

A place can have *activities* associated with it. Any user in the place can request to add activities to the place. Any user in the place can register to be informed of new activities added to the place. Activities define ways in which the users in the place can collaborate. Usually activities have a back-end piece on the Sametime server to support the activity. Such back-end activities are connected to the Places Service on the server using an *activity provider* included with the toolkit. (A future release of the toolkit will allow you to write your own utility provider.)

Place Members and Attributes

Users, sections, activities, and the place itself are all *place members*. Any place member can send messages to another place member. For example, a user can send a message to a specific activity or to a section, in which case all users in the section receive this message. The place member to which the message is sent defines the scope of the message. Place members also have *attributes* that they or other place members can manipulate according to the access controls enforced by the Places Service.

The Places Service

The Places Service of the toolkit contains objects that correspond to the virtual entities described above. The main objects are Place, Section, Activity, UserInPlace and MyselfInPlace (representing the currently logged in user in the place).

In this sample, the Places Service implements a meeting place where users can chat with each other. All users enter the stage and are therefore in the same section. You will use the message mechanism described previously to implement the chat in the meeting. All users send their chat messages to their section, and since all the users are in the same section in this sample, all the users in the place receive the chat messages.

The Chat Meeting Applet

The init() method of the meeting initializes a Sametime session object, loads and starts the Sametime components, registers a listener to login events, and finally requests to log in to the Sametime server with the login name and password passed as parameters to the applet.

Using the PlaceAwarenessList

The loggedIn() event, implemented by a CommunityEventsListener object, starts implementing the real behavior of the applet. When this event is generated, you call the layoutAppletUI() method, where you lay out all the AWT components used by the applet. These components include the chat panel (to be described in the following section), a label to display the number of users currently in the place, and a PlaceAwarenessList object to display the list of users in the meeting.

PlaceAwarenessList is an AWT component provided by the toolkit. It is very similar to the AwarenessList component used in previous samples to show the online status of a list of users in the community. The difference is that PlaceAwarenessList is used to show a dynamic list of users in a specific section of a place. It tracks the users as they enter and exit the section and shows the users with their latest online status.

Entering the Place

After calling the layoutAppletUI() method, the loggedIn() event handler calls the helper method enterPlace() to enter the place for the chat meeting. The enterPlace() method of the meeting applet gets a reference to the Places Service component from the session and requests that it create a place with the place name passed in the placeName parameter to the applet. After the place is created, a PlaceEventsListener object is added as a listener to the place. Finally, the enter() method of the created place object is called to actually enter the user into the place.

If the place is entered successfully, an entered() event is generated. The PlaceEventsListener object catches this event and calls the bindToSection() method of the PlaceAwarenessList object to connect it to the section the user has just entered. Finally, it calls the bindToPlace() method of the ChatPanel object to bind the chat panel to the place just entered.

Tracking Users Entering and Leaving the Place

The ParticipantListListener class inherits from AwarenessViewAdapter and implements the usersAdded() and usersRemoved() methods to track the number of people in the place and to display the correct number in the people number label of the applet. The implementation of these event handlers is straightforward.

Finally, the destroy() method of the applet is exactly the same as the implementation in previous samples. We log out, stop the session object, and unload it.

The Chat Panel

The Chat Panel provides a text field control and a button with which users can send text messages to all the participants in the place. It also provides a chat transcript area in which the entire transcript of the discussion in the place is shown. ChatPanel uses the ChatArea utility class included in the toolkit to implement the chat transcript functionality.

The ChatPanel constructor records the session and applet context and calls the init() method to lay out the chat area and the send panel, which consists of a send button and a TextField object. In the init() method, listeners are added to both the ChatArea object and to the TextField object.

The meeting applet calls the bindToPlace() method of the ChatPanel object once it receives the entered() event to let it know that it has successfully entered the meeting place. In this method, the ChatPanel object records the place it is bound to and adds a MyMsgEeventsListener to listen to message events sent to the user in the place.

How Text is Sent

After typing text in the text field, the user can send text to the place in two ways:

- **Click the Send button** – When the Send button was created in the init() method, the ChatPanel was added as an ActionListener to it. In the actionPerformed() event handler required by this listener, the text is taken from the text field, the text field is cleared, and the sendText() helper method is called to actually send the text. The sendText() helper method retrieves the section the user is currently in and sends the text to

that section. This code implicitly assumes that all users in the place are in the same section. After sending the text, focus is returned to the text field so that the user can type the next message.

Pressing the Enter key while the focus is in the text field – As mentioned previously, a listener was registered to the text field in the init() method. The class implementing this listener is TextFieldEventsListener that inherits from KeyAdapter. Only one event handler was implemented for this listener, keyPressed(), which is called when the user presses any key while the focus is in the text field. This handler simply verifies that the pressed key is the Enter key and then calls the sendText() method as in the previous case.

How Users Are Notified About Sent Text

As mentioned previously, the bindToPlace() method called by the applet registers a listener to receive message events sent to the user in the place. Messages in a place can be sent in different scopes: place, section, and user. Regardless of the scope, this listener receives them, if the user is the scope of the sent message. The class MyMsgEeventsListener extends the MyMsgAdapter class and implements a single event handler, textReceived(). This event handler is called if another place member sent text to the user, regardless of the scope in which it was sent. The implementation of this event handler ensures that the sender of the message is a user (an activity can also send messages to users in the place). It then checks to see if the sender of the text was the user himself or some other user. If the sender of the text was the user, the text is displayed in the chat area in a blue font; if the sender of the text was some other user, the text is displayed in the chat area in a black font.

In the init() method, a ChatAreaEventsListener object was added to the ChatArea object. This class implements only one event handler of ChatAreaAdapter, chatURLClicked(). This event handler is called when the user clicks on a URL in the chat area. The event object contains the URL that was clicked. In the sample implementation of this event, a new browser window is opened with this URL.

Appendix A. Deprecated AWT and Community UI Components

Previous versions of the Sametime Java Toolkit shipped with components that provided Sametime-specific UI behavior by extending Java AWT components. These Sametime-enabled AWT components have been deprecated starting with Sametime 7.5. Support for these will continue for two major releases at which point the components will be removed. User Interface technologies are continually evolving – the core Java Toolkit components can be used to achieve Sametime-enabled user interfaces on all of them including AWT, Swing, SWT, etc., so focus has been moved to the core components themselves (which the deprecated AWT components were built on). For more information on the core components, see the Community Services section above.

Many of the sections and samples in this tutorial reference these deprecated UI components. These samples still demonstrate valuable examples of how to use the core Java Toolkit components. The deprecated components are AWT-based APIs that listen to and provide UI wrappers to the core components. All references to the deprecated components will be removed in a future release. For more information on the core components, see the IBM Sametime *Java Toolkit Developer's Guide* and the Java Toolkit API JavaDoc.

The Toolkit UI

The toolkit provides a standard user interface (UI) that can be used by developers to speed up the development time required to build a Sametime-enabled application. The standard UI is provided either by toolkit UI components (similar to the toolkit components providing the different Community and Meeting services) or by AWT dialogs and panels, which can be embedded inside any AWT Container.

Community UI Components

The following table lists and describes the Community UI components available in the Sametime Java Toolkit.

Table listing community UI components.

Name of UI Component	Description
AnnouncementUI	This component provides the ability to send and receive announcements without having to deal with the low-level protocols that are involved. It includes the Send Announcement dialog used for writing and sending the announcement, and the Receive Announcement dialog for receiving announcements.
ChatUI	This component provides standard UI to create and receive instant messages and chat meetings with two or more participants. This standard UI can be customized by the developer.
CommUI	This component provides a standard UI to display community messages such as logout and administrator messages. It also provides name resolving with UI.
FileTransferUI	This component provides the ability to send and receive files without having to deal with the low-level protocols that are involved. It includes the Send File dialog used for choosing the file to send and initiating the file transfer, the Receive File dialog for accepting or declining the file transfer, and the File Transfer Status dialog that indicates the progress of the transfer on both sides.

Community AWT Dialogs and Panels

The following table lists and describes the Sametime-enabled AWT components available in the Sametime Java Toolkit.

Table listing Sametime-enabled AWT components.

Name of AWT Dialog or Panel	Description
Awareness List	This embeddable AWT panel displays a list of users in the community and their current online status. You can launch instant messages and Sametime meetings with participants from the list by selecting one or more users and selecting the required type of meeting from the context menu.
Place Awareness List	This embeddable AWT panel displays a list of users in the community that are currently in a specific place. The users current online status are shown. You can launch instant messages and Sametime meetings with participants from the list by selecting one or more users and selecting the required type of meeting from the context menu.
Capabilities List	This embeddable AWT panel displays the audio/video tools of selected users.
Tools Dialog	This dialog displays the audio/video tools of selected users.
Privacy Dialog and Panel	This dialog and panel provide a standard UI for setting and viewing your Sametime privacy settings.
Add Dialog	This dialog is used to select users in the community by entering their names or browsing for them in the Sametime directory.
Resolve Panel	This embeddable AWT panel is used to resolve user names.
Directory Dialog and Panel	This dialog and panel provide a standard UI for browsing and searching the Sametime directory for users and groups.
Group Content Dialog	This dialog displays the content of a Sametime directory group.

Appendix B. Deprecated Meeting Services APIs

Previous versions of the Sametime Java Toolkit shipped with Meeting Services Java APIs which allow client Java applets and applications to create and join multi-participant meetings. These Meeting Services Java APIs have been deprecated starting with Sametime 7.5. Support for these APIs will continue for two major releases.

The APIs are used to share and annotate documents, share and control applications, and communicate using live audio and video between meeting participants. The developer can choose the tools to use in each meeting and control the visual layout and containment of various UI components. The Meeting Services include Application Sharing, Shared Whiteboard, and Interactive Audio and Video and provides a highly scalable and lightweight Broadcast Receiver. The Broadcast Receiver receives and interprets low bandwidth audio, video, and data streams generated by the different Meeting Services, and displays these without actively participating in the meeting.

The samples and documentation relating to these deprecated APIs have been removed from this tutorial, but the Meeting Service API libraries along with full documentation and samples are included in the Sametime 7.0 Java Toolkit which can be downloaded from the IBM developerWorks Lotus downloads page:

http://www-128.ibm.com/developerworks/lotus/downloads/toolkits.html

Navigate to the IBM Sametime section, click on Sametime Java toolkit, then select IBM Sametime 7.0 Java Toolkit to download the 7.0 version which includes all the Meeting Services materials.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
5 Technology Park Drive
Westford Technology Park
Westford, MA 01886

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp.
Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

These terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM

AIX

DB2

DB2 Universal Database Domino

Domino

Domino Designer

Domino Directory

i5/OS

iSeries

Lotus

Notes

OS/400

Sametime

System i

WebSphere

AOL is a registered trademark of AOL LLC in the United States, other countries, or both.

AOL Instant Messenger is a trademark of AOL LLC in the United States, other countries, or both.

Google Talk is a trademark of Google, Inc, in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, and Windows are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.