

UltraLightClient Essentials Guide

UltraLightClient '08 Update 4

canoo

Canoo Engineering AG
Kirschgartenstrasse 5
CH-4051 Basel
Switzerland
Tel: +41 61 228 9444
Fax: +41 61 228 9449
ulc-info@canoo.com
<http://www.canoo.com/ulc/>

Copyright 2000-2009 Canoo Engineering AG
All Rights Reserved.

DISCLAIMER OF WARRANTIES

This document is provided “as is”. This document could include technical inaccuracies or typographical errors. This document is provided for informational purposes only, and the information herein is subject to change without notice. Please report any errors herein to Canoo Engineering AG. Canoo Engineering AG does not provide any warranties covering and specifically disclaim any liability in connection with this document.

TRADEMARKS

Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc.

IBM and WebSphere are trademarks or registered trademarks of International Business Machines Corporation.

All other trademarks referenced herein are trademarks or registered trademarks of their respective holders.

Contents

2.1	Architecture	4
2.2	Usage	5
2.2.1	Programmatic Configuration.....	5
2.2.2	Configuration Using Program Arguments.....	6
2.2.3	Graphical User Interface	8
3.1	Application	10
3.1.1	IApplication	10
3.1.2	AbstractApplication	11
3.2	Infrastructure	11
3.2.1	ApplicationContext	11
4.1	Reading and Writing Files	13
4.1.1	Choosing a File Located on the Client Machine.....	13
4.1.2	Reading from a File Located on the Client Machine.....	14
4.1.3	Writing to a File Located on the Client Machine.....	15
4.2	Showing a Document in a Web Browser	17
4.3	Accessing the Client Environment.....	17
4.4	Accessing the Client's UI Defaults.....	18
4.5	Configuring Communication	19
4.5.1	Event Delivery Modes	20
4.5.2	Model Update Modes.....	20
5.1	Servlet Container Access	22
5.2	EJB Container Access.....	23
6.1	Logging Classes	24
6.2	Configuring the Client-Side (UI Engine) Log Manager.....	24
6.3	Configuring the Server-Side (ULC) Log Manager.....	25
7.1	General Concepts.....	27
7.2	Using ULCBoxPane.....	28
7.2.1	Planning the Layout	28
7.2.2	A Simple Address Form Using Nested Boxes.....	29
7.2.3	A Simple Address Form Using Spanning	31
8.1	Drag & Drop Operation Overview	32
8.2	Configuration	32
8.3	Transfer Data	33
8.3.1	DataFlavor	34
8.3.2	IDnDData	34
8.4	TransferHandler.....	35
9.1	Serializability	38
9.2	Programming Restrictions	39
10.1	Example showing Pause and Resume Functionality.....	40
10.1.1	Launcher Implementation.....	40
10.1.2	Server-Side Implementation.....	42

1 Overview

UltraLightClient is a library to build Rich Internet Applications (RIA) in Java. Use this standard Java library to develop rich, responsive graphical user interfaces (GUIs) for enterprise web applications within Java EE and Java SE infrastructures.

This edition of the *UltraLightClient Essentials Guide* accompanies UltraLightClient '08. It provides useful tips and practices to develop with ULC. You may also want to refer to the [ULC Reference Guide](#), which provides detailed descriptions of ULC classes for standard application development. Moreover the [ULC Extension Guide](#) discusses how ULC widget set can be extended.

It is assumed that the reader is familiar with Java programming and the basics of the Java Swing widget set. Since ULC applications are pure Java applications, they can be built using any suitable Java 2-compatible development environment (JRE 1.4 or higher).

Organization

This document is organized into the following chapters:

Chapter 2 provides information about the development environment.

Chapter 3 describes the application main classes.

Chapter 4 describes client access.

Chapter 5 discusses container access.

Chapter 6 provides a description how to set up logging.

Chapter 7 explains ULC layout concepts and the *ULCBoxPane* layout.

Chapter 8 describes ULC Drag and Drop mechanism.

Chapter 9 discusses EJB development.

Chapter 10 describes the *Pause()/Resume()* API.

Chapter 11 lists best practices and issues to avoid when developing ULC applications

2 Development Environment

This chapter describes how to configure the development environment and set up the *DevelopmentRunner*.

ULC applications and clients can be deployed in various ways; the server part can be run in a servlet container or an EJB container (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)), the client part can be run standalone, as an applet, or using a JNLP client such as Java Web Start. In a production environment the client and server parts will typically run on different computers.

At development time, however, application developers typically like to run everything locally in order to keep the deployment process as lightweight as possible. The ULC development environment package allows both client and server to run within the same Java Virtual Machine (JVM).

2.1 Architecture

The class `com.ulcjava.base.development.DevelopmentRunner` provides a runtime environment for the client as well as for the server part. The *DevelopmentRunner* first starts the server, and then connects the client to it. Both client and server parts run inside the same virtual machine.

As with all other deployment scenarios, server and client half objects communicate through the ULC communication infrastructure. Since the two parts run in the same JVM, the communication can be implemented in a straightforward way by the ULC framework where the ULC requests between client and the server are exchanged using request queues.

The development environment package provides ways to simulate all relevant deployment and runtime aspects that a ULC application developer must be aware of, such as *init* and *user* parameters, and connection characteristics. For convenience, a graphical user interface is provided that can be used to control these aspects. Moreover, this user interface provides the means to investigate the communication between the server and client half objects.

The *DevelopmentRunner* is suited to provide a ULC developer with early feedback on how an application will perform under expected network conditions. Note that the applet module provides an *AppletDevelopmentRunner* that should be used whenever a ULC application uses the *ULCAppletPane* component. It is based on the standard *DevelopmentRunner* and displays the applet pane in a separate applet viewer window.

2.2 Usage

The developer can configure the following parameters when using the *DevelopmentRunner*:

Parameter	Description
Application main class	The application main class implementing the <i>com.ulcjava.base.application.IApplication</i> interface (see Section 3.1.1).
Init parameters	The init parameters that are provided to the application (see Section 3.2.1). At deployment time these parameters are specified through (server-side) container configuration.
User parameters	The user parameters that are passed to the application (see Section 4.3). At deployment time these user parameters are provided by the (client-side) launcher component.
Connection characteristics	Connection characteristics used for simulation of transport delays. A range of typical configurations is provided. Additionally, the developer can define custom configurations.
Log level	Log level used on this application. The developer can change the log level used in the development environment.

To configure these parameters, the *DevelopmentRunner* class provides three usage modes that are described in the following sections:

- The *DevelopmentRunner* can be configured programmatically using setter methods.
- The *DevelopmentRunner* can be configured using program startup parameters.
- The developer can use a graphical user interface that also provides a monitor visualizing information on client-server communication. Moreover this user interface can be used to pause and resume the application and to test passivation and activation of application sessions.

2.2.1 Programmatic Configuration

The *DevelopmentRunner* class provides the following static methods to programmatically configure and start it:

Method	Description
<i>setApplicationClass(Class)</i>	Sets the application main class implementing the <i>com.ulcjava.base.application.IApplication</i> interface.
<i>setInitParameters(Properties)</i>	Sets the init parameters as defined by the key-value pairs in the supplied <i>java.util.Properties</i> object.
<i>setUserParameters(Properties)</i>	Sets the user parameters as defined by the key-

Method	Description
	value pairs in the supplied <i>java.util.Properties</i> object.
<i>setConnectionType(ConnectionType)</i>	Sets the connection type used for the simulation of transport delays to the supplied <i>com.ulcjava.base.development.ConnectionType</i> object.
<i>setLogLevel(Level)</i>	Sets the log level of the application. The argument has to be an instance of <i>com.ulcjava.base.shared.logging.Level</i> . The default is set to <i>com.ulcjava.base.shared.logging.Level.WARNING</i> .
<i>setUseGui(Boolean)</i>	If set to <i>true</i> , the graphical user interface of the <i>DevelopmentRunner</i> will be opened when <i>run</i> is called (see Section 2.2.3).
<i>run()</i>	Starts the <i>DevelopmentRunner</i> using the currently defined configuration.

Typically you will need to create a development launcher class which configures and starts the *DevelopmentRunner* for your specific application. This is done in the *main()* method of the launcher. The following code shows a typical example of such a development launcher.

```
public class TeamMembersDevelopmentLauncher {
    public static void main(String[] args) throws Exception {
        DevelopmentRunner.setApplicationClass(TeamMembers.class);
        DevelopmentRunner.run();
    }
}
```

2.2.2 Configuration Using Program Arguments

In addition to the methods for programmatic configuration, the *DevelopmentRunner* class offers a *main()* method to support passing program arguments:

Program argument	Description	Required
-applicationClass <i>application class</i>	The fully qualified application class name (e.g., <i>com.ulcjava.sample.hello.Hello</i>).	Yes Optional if <i>-useGui</i> is specified.
-initParameter <i>key=value</i>	Defines an init parameter. The format of the key-value argument must be as in Java property files.	No May be repeated.
-userParameter <i>key=value</i>	Defines a user parameter. The format of the key-value argument must be as in Java property files.	No May be repeated.

Program argument	Description	Required
-connectionType <i>connection type</i>	Defines the connection type to be used. Must be one of the following: UNLIMITED, LAN10M, LAN1M, DSL512K, DSL256K, DSL112K, MODEM56K, MODEM28K, or custom.	No Default is UNLIMITED
-logLevel <i>logLevelName</i>	Sets the log level.	No Default is WARNING.
-useGui	If present, the graphical user interface will be opened (see 2.2.3: Graphical User Interface).	No
-reloadClasses (only applicable together with the -useGui option)	If present, classes will be reloaded when the Start button in the GUI is pressed. Classes to be excluded from reloading can be defined in a file <i>excluded.properties</i> , located anywhere on the classpath (see 2.2.3: Graphical User Interface).	No

The following command line shows how to start the *Hello* application using the *DevelopmentRunner*; additionally two init parameters are defined and the connection type is configured to be a 28K modem connection.

```
java com.ulcjava.base.development.DevelopmentRunner
  -applicationClass com.ulcjava.sample.hello.Hello
  -initParameter db-url=jdbc:mysql://somehost/test
  -initParameter mail-host=someotherhost
  -connectionType MODEM56K
```

Programmatic configuration and configuration using program arguments may be combined as illustrated by the following example. The example defines the application main class programmatically. All other parameters are taken from the program parameters. Note that program arguments will overwrite programmatic configuration if there are conflicts.

```
public class TeamMembersDevelopmentLauncher {
    public static void main(String[] args) throws Exception {
        DevelopmentRunner.setApplicationClass(TeamMembers.class);
        DevelopmentRunner.main(args);
    }
}
```

2.2.3 Graphical User Interface

The graphical user interface (GUI) of the *DevelopmentRunner* can be used to conveniently enter all configuration parameters. Using the provided controls, an application session can be started and stopped from this GUI. Moreover, an application session can be paused and resumed as well as passivated and activated.

In addition, the GUI provides information on the communication between client and server and offers the possibility to dump ULC requests to a console for debugging purposes.

When the **Class reloading** checkbox is selected, all application classes are reloaded during the next startup of the application. To reload, the *DevelopmentRunner* uses a custom class loader that sits in front of the system class loader. Classes in the *com.ulcjava.base* packages are always excluded from class reloading. In addition, classes defined in an *excluded.properties* file are loaded by the system class loader and are hence also excluded from class reloading. Class and package entries in this file must obey the following syntax:

```
excluded.<nr>=<package or class name>
```

Example:

```
excluded.0=org.*
excluded.1=com.weblogic.*
excluded.2=com.acme.StartupClass
excluded.3=com.oracle.*
```

There is a default *excluded.properties* file located in the *com.ulcjava.base.development* package that contains the most common classes that need to be loaded by the system class loader. Place a custom *excluded.properties* file on the *DevelopmentRunner* classpath before the default, either in the same package or in the default package.

Figure 1 shows a screenshot of the *DevelopmentRunner* GUI.

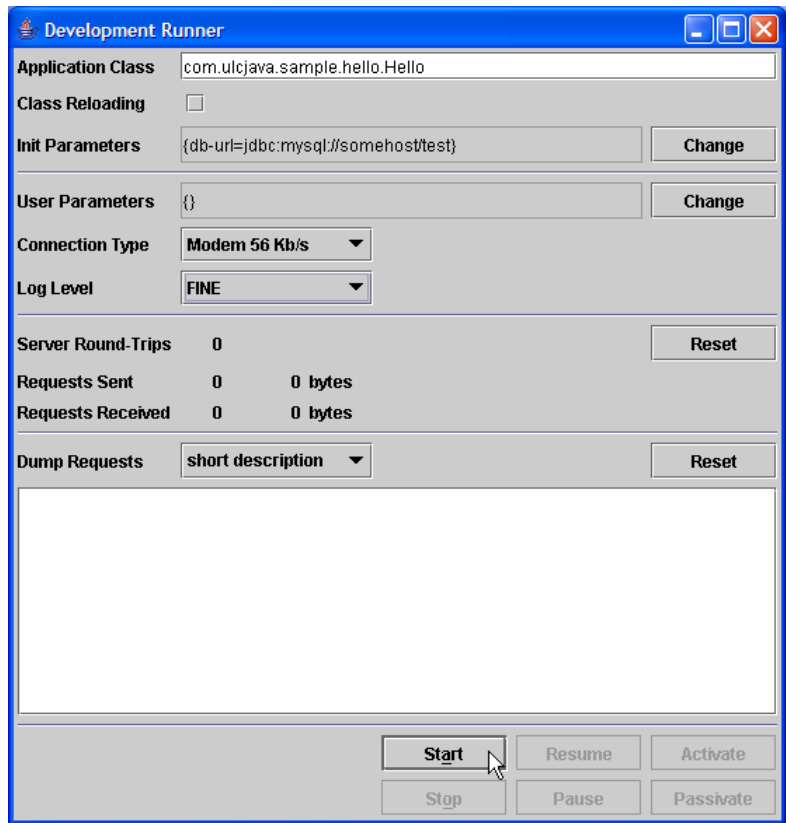


Figure 1: Development Runner

The graphical user interface is started in either of the two ways described in Sections 2.2.1 and 2.2.2. Init and user parameters that are already defined at GUI startup can be changed before the application is started.

3 Application Main Classes

The following sections discuss how to create a containment hierarchy for ULC widgets and add widgets and other containers to it. Additionally, examples are given that show the usage of the various models and widgets.

Please note that in all the examples any text passed to *System.out.println()* will be written to the server-side console.

The next section describes the main classes that are used to implement a ULC application.

3.1 Application

Implementing the *com.ulcjava.base.application.IApplication* interface creates a ULC application. The *IApplication* interface defines the methods that control the lifecycle of a ULC application.

In accordance with the long tradition of developers around the world, let us start ULC programming by writing the classic “Hello World” application. The following code shows a fully functional ULC application that displays a frame containing a single label. Additionally, the application is configured to terminate when the user closes the frame.

```
public class Hello extends AbstractApplication {
    public void start() {
        ULCFRAME frame = new ULCFRAME("Hello");
        frame.setDefaultCloseOperation(ULCFRAME.TERMINATE_ON_CLOSE);
        frame.add(new ULCLABEL("Hello world!"));
        frame.setVisible(true);
    }
}
```

The following picture shows a screenshot of the running application. Note that this application is also provided as a sample in the samples module of this release (see *Sample Module* in the [ULC Installation Guide](#)).

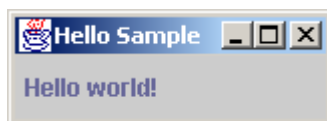


Figure 2: Hello World sample application

For more technical information on how applications are created and managed, please see the chapter on *Server Architecture* in the [ULC Architecture Guide](#).

3.1.1 IApplication

The *com.ulcjava.base.application.IApplication* interface is the central interface that ULC applications must implement. For every connecting client, a new instance of the class implementing this interface is created by the underlying runtime container adapter (e.g., servlet container adapter). For more information about sharing components see the section on *Best Practices* in the [ULC Essentials Guide](#).

Features

- The *start()* method is the first method that is called after a new instance of the implementation has been created. Applications build up their first view within this method and make the view visible.
- The *stop()* method is the last method that is called before a session is terminated. Applications typically use this method to clean up and release any resources used for this session.
- The *passivate()* method is called before the session is passivated. This method only needs to be implemented for applications that are deployed in server containers that support session passivation (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)). In this method the application is supposed to release any resources (e.g., DB connections) or transient data that should not be serialized.
- The *activate()* method is called after the server session has been activated again. It must be implemented for applications that are deployed in server containers that support passivation, *activate()* should reopen resources or restore transient state.
- The *handleMessage()* method is called when the server session receives a message from the client session. This mechanism can be used when implementing custom client launcher classes to notify the application about special client-side events (see the example in the chapter on *Interacting with the Enclosing HTML Page* in the [ULC Deployment Guide](#))

See Also

AbstractApplication

3.1.2 AbstractApplication

The *com.ulcjava.base.application.AbstractApplication* class is a default implementation of the *com.ulcjava.base.application.IApplication* interface.

Features

- The *AbstractApplication* class provides empty implementations of all methods but the *start()* method. Many ULC applications will only need to implement the *start()* method.

See Also

ApplicationContext, IApplication

3.2 Infrastructure

3.2.1 ApplicationContext

The *com.ulcjava.base.application.ApplicationContext* class provides access to the application's runtime environment. ULC applications can use this class to access the environment in a container-independent way. Note that all methods are static, so the ULC framework forwards the call to the corresponding application instance.

Features

- The *terminate()* method is used to terminate the current session. A call to this method will close all open windows, call the *stop()* method on the *IApplication* implementation, and terminate the session.
- The *getApplication()* method returns the actual *IApplication* instance. This method may be used to easily access the main application instance from anywhere in the application code.
- The *setAttribute()* and *getAttribute()* methods may be used to easily access objects from anywhere in the application code. The *getAttributeNames()* method returns the names of all attributes of the ULC application. The *removeAttribute()* method removes an attribute from the ULC application.
- The *getInitParameter()* method returns the value of the named initialization parameter as defined by the deployment. The *getInitParameterNames()* method returns the names of all initialization parameters of the ULC application. These calls are forwarded to the underlying runtime container (e.g., Servlet container), the actual declaration of these init parameters at deployment time is therefore container-dependent (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)).
- The *getUserPrincipal()* method returns a *java.security.Principal* object containing the name of the current authenticated user. If the user has not been authenticated, the method returns *null*. A call to this method is forwarded to the underlying runtime container, the setup of the authentication configuration is container-dependent (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)).
- The *isUserInRole()* method returns a boolean indicating whether the authenticated user is included in the specified logical role. If the user has not been authenticated, the method returns *false*. A call to this method is forwarded to the underlying runtime container, definition of roles and role membership is container-dependent (see the chapter on *Server Deployment* in the [ULC Deployment Guide](#)).

Examples

The following code snippet shows how to use the *ApplicationContext* to terminate the application.

```
quitMenuItem.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event) {
        ApplicationContext.terminate();
    }
});
```

The following example demonstrates how to access init parameters defined at deployment time.

```
String dbUrl = ApplicationContext.getInitParameter("db-url");
```

See Also

IApplication, *AbstractApplication*

4 Client Access

This chapter describes how to access the client machine. The necessary methods are provided as static members of the *ClientContext* class.

4.1 Reading and Writing Files

4.1.1 Choosing a File Located on the Client Machine

Choosing a file located on the client machine is a two-step operation. First the static method *chooseFile()* in the *ClientContext* class has to be called. This method has to be provided with an *IFileChooseHandler* argument. In a second step the ULC framework invokes a hook in the provided *IFileChooseHandler* when the selection has been completed.

Features

- The *IFileChooseHandler* interface provides hooks for successful selections and for error handling.
- The user chooses the file with a file chooser dialog that can be configured with a *FileChooserConfig* object.
- When choosing a file with a file chooser dialog, a *ULCComponent* is configured as the parent for the file chooser dialog
- The chosen filename is provided by the on success hook of the *IFileChooseHandler*

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

Examples

In this example the application asks the user to choose a text file located on the client machine. The only selectable files are those with extension ".txt".

```
// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a text file");
fcConfig.addFileFilterConfig(new FileChooserConfig.FileFilterConfig(
    new String[]{"*.txt"}, "text files (*.txt)"));

ClientContext.chooseFile(new IFileChooseHandler() {
    public void onSuccess(String filePath) {
        System.out.println("Chosen text file is: " + filePath);
        // return the file path
    }

    public void onFailure(int reason, String description) {
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);
```

In the next example the application asks the user to choose a directory (only directories are shown).

```
fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a directory");
fcConfig.setFileSelectionMode(FileChooserConfig.DIRECTORIES_ONLY);

ClientContext.chooseFile(new IFileChooseHandler() {
    public void onSuccess(String filePath) {
        System.out.println("Chosen directory is: " + filePath);
        // return the file path
    }

    public void onFailure(int reason, String description) {
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);
```

4.1.2 Reading from a File Located on the Client Machine

Reading from a file located on the client machine is a two-step operation. The first step is to initiate the client file download by calling the static *loadFile()* method in the *ClientContext* class. This method has to be provided with an *IFileLoadHandler* argument. In a second step the ULC framework invokes a hook in the provided *IFileLoadHandler* when the download has been completed.

Features

- The *IFileLoadHandler* interface provides hooks for successful downloads and for error handling. For successful downloads the file data can be accessed by means of a provided input stream.
- The file to be downloaded can be specified as a client file pathname. Alternatively, the application lets the user choose the file with a file chooser dialog that is configured with a *FileChooserConfig* object. With a file chooser dialog only the file selection mode *FileChooserConfig.FILES_ONLY* is allowed.

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

Examples

In the following example the application attempts to load a property file located on the client machine.

```
ClientContext.loadFile(new IFileLoadHandler() {
    public void onSuccess(InputStream in, String filePath) {
        Properties p = new Properties();
        try {
            p.load(in);
        } catch (IOException e) {
        }
    }
});
```

```

        // process properties
    }

    public void onFailure(int reason, String description) {
        // ignored - the default properties will be used
    }
}, "C:\\myapp\\myapp.properties");

```

In the next example the application asks the user to choose a file containing some report. The only selectable files are those with extension ".rep".

```

// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Choose a report to be loaded");
fcConfig.addFileFilterConfig(new
    FileChooserConfig.FileFilterConfig(new String[]{" .rep"},
        "report files (*.rep)"));

ClientContext.loadFile(new IFileLoadHandler() {
    public void onSuccess(InputStream in, String filePath) {
        // read the report from the input stream and process it
    }

    public void onFailure(int reason, String description) {
        // show an alert informing the operation has failed
    }
}, fcConfig, frame);

```

4.1.3 Writing to a File Located on the Client Machine

Writing to a file located on the client machine is a three-step operation. The first step is to initiate the client file upload by calling the static *storeFile()* method in the *ClientContext* class. This method has to be provided with an *IFileStoreHandler* argument. To collect the data to be uploaded, the ULC framework invokes a hook in the provided *IFileStoreHandler* (step 2). Finally, another hook in *IFileStoreHandler* is called when the upload has finished.

Features

- The data to be uploaded has to be written to an output stream in the *prepareFile()* hook of the *IFileStoreHandler*. The method *prepareFile()* may throw an exception if the data cannot be written. In this case, the upload process will be terminated.
- The *IFileStoreHandler* interface provides hooks for successful uploads and for error handling.
- The file to be written to can be specified as a client file pathname. Alternatively the application can let the user choose the file with a configurable file chooser dialog.

Not all client environments support client-side file operations. For instance, applets running in the default applet sandbox are not allowed to read and write files.

Examples

In the following example the application attempts to save a property file on the client machine:

```
try {
    ClientContext.storeFile(new IFileStoreHandler() {
        public void prepareFile(OutputStream data) throws Exception {
            fProperties.store(data, "myapplication properties");
        }

        public void onSuccess(String filePath) {
            // bring up an alert reporting the success
        }

        public void onFailure(int reason, String description) {
            // bring up an alert reporting the failure
        }
    }, "C:\\myapp\\myapp.properties");
} catch {Exception e}
    // handle any exceptions that may have occurred, e.g., in prepareFile()
}
```

In the next example the application asks the user to choose a place and filename under which an exported report will be saved:

```
// parent for the file chooser
ULCFrame frame = new ULCFrame();
// ...

FileChooserConfig fcConfig = new FileChooserConfig();

fcConfig.setDialogTitle("Where would you like to save the report?");

try {
    ClientContext.storeFile(new IFileStoreHandler() {
        public void prepareFile(OutputStream data) throws Exception {
            // write report data to the 'data' stream
        }

        public void onSuccess(String filePath) {
            // bring up an alert reporting the success
        }

        public void onFailure(int reason, String description) {
            // bring up an alert reporting the failure
        }
    }, fcConfig, frame);
} catch {Exception e}
    // handle any exceptions that may have occurred, e.g., in prepareFile()
```

```
}
```

4.2 Showing a Document in a Web Browser

Showing a given URL in a web browser can be achieved by calling the static method *ClientContext.showDocument()*. How the web browser is accessed and what URLs are accepted depends on the configuration of the client environment, i.e., on the platform and on the installed browser service (see *Browser Service* in the [ULC Architecture Guide](#)). Using the *AppletBrowserService* or the *JnlpBrowserService*, for instance, only valid protocols (i.e., protocols for which there is a protocol handler installed) are allowed in the document's URL.

Irrespective of the installed browser service, the security manager installed on the client (e.g., a jnlp or applet security manager) might disallow security-critical operations such as reading local files.

Features

- Shows a given URL in a web browser.
- One variant of the *showDocument()* method defines the frame target name as defined by [6]. This target argument is only interpreted in the case of the UI Engine running as an applet; it is ignored otherwise.
- It is not possible to track whether displaying a document in the browser was successful or not.
- On Windows platforms, files are displayed outside the browser when the *StandaloneBrowserService* is installed and either no protocol or the *file* protocol is used in the document's URL. The security manager, however, might disallow this operation.

Examples

The following example outlines the usage of *ClientContext* to display a document in a web browser.

```
final ULCTextField urlField = new ULCTextField(30);

urlField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ClientContext.showDocument(urlField.getText());
    }
});
```

4.3 Accessing the Client Environment

The *ClientContext* class also provides methods to access information about the client system environment.

Features

- *getAddress()* returns the client IP address

-
- `getAvailableFontFamilyNames()` returns an array containing the names of all font families.
 - `getHost()` returns the client host name
 - `getLocale()` returns the client locale
 - `getLookAndFeelName()` returns the client's look and feel name
 - `getScreenHeight()`, `getScreenResolution()`, `getScreenWidth()` return the client screen attributes
 - `getSystemColor(String key)` returns the system color for the specified key. See the `SystemColor` class for all valid keys.
 - `getSystemPropertyNames()` returns an array containing the names of all client system properties, use `getSystemProperty(String)` to retrieve individual values.
 - `getTimeZone()` returns the client time zone.
 - `getUserParameterNames()` returns an array containing the names of all user parameters, use `getUserParameter(String)` to retrieve individual values.

Examples

ULC applications can be easily localized by using the standard Java internationalization support. The resource bundle is retrieved based upon the connecting client's *Locale* object:

```
Locale clientLocale = ClientContext.getLocale();
ResourceBundle myResources =
    ResourceBundle.getBundle("MyResources", clientLocale);

//...

ULCButton okButton =
    new ULCButton(myResources.getString("okButtonLabel"));
```

4.4 Accessing the Client's UI Defaults

It is possible for ULC applications to access the client's most important UI defaults for which there is no API available (in particular, all default borders, colors, fonts, insets, and integer-valued properties). Access to these default values is especially valuable when creating customized cell renderers for tables, trees, etc. but they also prove useful for adjusting and fine-tuning an application's appearance. Note that a ULC application cannot change these defaults.

Initially, i.e., as soon as the connection from the client to the ULC application is established, all colors, fonts, insets, and integer-valued UI defaults are transferred to the application. In this way, they are immediately available to the application and can already be used for the very first user interface window. Therefore, any changes to the UI defaults have to be made by the launcher *before* the connection is initiated.

Features

- `getBorder(String key)` returns the UI default border for the specified key.
- `getColor(String key)` returns the UI default color for the specified key.

- `getFont(String key)` returns the UI default font for the specified key.
- `getInsets(String key)` returns the UI default insets for the specified key.
- `getInt(String key)` returns the UI default integer values for the specified key.

The key is made up of the property prefix defined in the swing UI classes and the property name separated with a dot (*propertyPrefix* + . + *propertyName*, e.g. *CheckBox.foreground*), or just property name if there is no component name (e.g. *activeCaption*).

Examples

In the following code snippet, a custom renderer is installed on a *ULCComboBox*. A label is used as the rendering component to right-align the *Double* values in the drop-down list. However, since a label does not change its background and foreground colors when it is selected, we set these values explicitly according to the client's default selection foreground and background colors of *ULCComboBox*. There is no API available on the *ULCComboBox* to retrieve these default values, therefore we use the static methods in the *ClientContext* class.

```

Double[] values = getAvailableValues();
ULCComboBox comboBox = new ULCComboBox(values);
comboBox.setRenderer(new NumberRenderer());

...

private static class NumberRenderer implements IComboBoxCellRenderer {
    private ULCLabel fLabel;

    public NumberRenderer() {
        fLabel = new ULCLabel();
        fLabel.setHorizontalAlignment(ULCLabel.RIGHT);
    }

    public IRendererComponent getComboBoxCellRendererComponent(
        ULCComboBox comboBox, Object value, boolean isSelected, int row) {

        fLabel.setForeground(isSelected ?
            ClientContext.getColor("ComboBox.selectionForeground") :
            comboBox.getForeground());
        fLabel.setBackground(isSelected ?
            ClientContext.getColor("ComboBox.selectionBackground") :
            comboBox.getBackground());
        return fLabel;
    }
}

```

4.5 Configuring Communication

Due to the distributed nature of ULC applications, event delivery and data updates from client to server may take a considerable amount of time. For this reason, ULC minimizes network traffic whenever possible. It also keeps the user interface as responsive as possible, e.g., dynamic loading of data does not block the user interface.

The *ClientContext* class provides an API to further configure the communication behavior of an application's widgets and models.

4.5.1 Event Delivery Modes

When sending events, the user interface must be blocked to ensure state consistency between client and server. The ULC framework does not completely block the client (e.g., repainting of the user interface is still performed) but merely blocks any user input (either by mouse or keyboard) until the server roundtrip triggered by the event has been completed and all side effects on the server and the respective updates on the client have been accomplished. In most situations, the user will not be aware of the fact that the user interface has been blocked for a short period of time.

To further improve the usability of a user interface, ULC widgets can be configured to deliver certain events in an asynchronous way. With such a configuration, users are able to continue to interact with the user interface while the event is being delivered to the application. Appropriate usage of this feature can significantly improve the usability. However, the application developer must ensure consistency by application-specific means.

Features

- Using the *ClientContext.setEventDeliveryMode()* method, events of a specific type and for a specific component can be configured to be delivered synchronously (i.e., the user interface is blocked) or asynchronously (i.e., the user can continue to interact with the user interface). By default, events are delivered synchronously.

Examples

The following example shows how to register an *ISelectionChangedListener* on the *ULCListSelectionModel* of a *ULCTable* and how to configure the *ULCListSelectionModel* to deliver *ListSelectionEvents* asynchronously to the server side.

```
table.getSelectionModel().addListSelectionListener(new StatusBarUpdater());
ClientContext.setEventDeliveryMode(table.getSelectionModel(),
    UlcEventCategories.LIST_SELECTION_EVENT_CATEGORY,
    UlcEventConstants.ASYNCHRONOUS_MODE);
```

4.5.2 Model Update Modes

Whenever tables, table trees, and trees are configured to provide in-place editing, the corresponding models (*ITableModel*, *ITableTreeModel*, and *ITreeModel*) are updated by default in *UlcEventConstants.DEFERRED_MODE* i.e., the update is sent with the next event. To send the model update event immediately, choose *UlcEventConstants.SYNCHRONOUS_MODE*. In order to minimize network traffic *UlcEventConstants.DEFERRED_MODE* has been made default for the above model updates, i.e., these models are updated only when other events are sent to the server side (e.g., when the user clicks a button). When the user clicks on a button, the model is first updated, and then the application's action listeners are called..

Features

- Using the *ClientContext.setModelUpdateMode()* method, model updates for a specific model can be configured to be delivered to the server immediately or deferred (i.e., the update is sent with the next event).

Examples

The following example configures an *ITableModel* to be updated immediately after the user edits the corresponding table.

```
ClientContext.setModelUpdateMode(table.getModel(),  
                                UlcEventConstants.SYNCHRONOUS_MODE);
```

5 Container Access

The *ApplicationContext* class provides an API to access container runtime information that is independent of the actual underlying container that the ULC application is running in (see Section 3.2.1).

In addition to these standard services, a specific implementation of a container adapter may offer additional services and access to container specific information.

Important note: A standard ULC application should not use container specific APIs, only applications with special requirements are supposed to do so. As these APIs are limited to applications being deployed and run in specific containers, you must be aware that using this API restricts deployment to such containers only. Also, an in-depth knowledge of the underlying container and ULC architecture is required, as incorrect manipulation of the objects returned by this API may interfere with the ULC runtime system.

5.1 Servlet Container Access

The ULC servlet container adapter offers an API to access servlet container specific runtime information. For instance, this API can be used to access the underlying *HttpSession* object that a ULC application session resides in. Using this approach, HTML-ULC multichannel applications can be written to present the same session data simultaneously in the ULC application client and in a web browser.

Features

The *com.ulcjava.container.servlet.application.ServletContainerContext* class provides access to the following information:

- The *javax.servlet.http.HttpSession* object the ULC application session resides in.
- The *javax.servlet.ServletContext* object corresponding to the web application that the ULC servlet container adapter was configured to be part of.
- The HTTP servlet request and response object of the current server roundtrip.

Examples

The following example shows a servlet displaying a session attribute and a ULC application that changes the same attribute directly in the *HttpSession* object. At deployment time it must be ensured that both the servlet and the ULC application (i.e., the servlet container adapter servlet) share the same session. More specifically, they must be part of the same web application and it must be ensured that session tracking information is sustained between the UI Engine (running as applet) connecting to the servlet engine and the servlet rendering the HTML page.

The displaying servlet can be implemented as follows:

```
public class MultiChannelServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
    }
}
```

```

        out.println("Text: " +
            httpRequest.getSession().getAttribute("text"));
        out.println("</BODY></HTML>");
    }
}

```

Using the *ServletContainerContext* class, the ULC application can directly access the attributes of the *HttpSession* that the application is running in:

```

public class MultiChannelApplication extends AbstractApplication {
    private ULCTextField fField;

    public void start() {
        ULCAppletPane contentPane = ULCAppletPane.getInstance();
        ULCBoxPane boxPane = new ULCBoxPane(true);

        String text = (String)ServletContainerContext.
            getHttpSession().getAttribute("text");
        fField = new ULCTextField(text, 20);
        boxPane.add(fField);

        ULCButton saveButton = new ULCButton("Save");
        saveButton.addActionListener(new IActionListener() {
            public void actionPerformed(ActionEvent event) {
                ServletContainerContext.getHttpSession().
                    setAttribute("text", fField.getText());
            }
        });
        boxPane.add(saveButton);

        contentPane.add(boxPane);
        contentPane.setVisible(true);
    }
}

```

5.2 EJB Container Access

The EJB container adapter implementation currently does not offer additional container specific services. Future releases may add such functionality.

6 Setting up Logging for ULC

This section explains the logging information provided by ULC. In addition, a description on how to configure the log manager is provided.

6.1 Logging Classes

ULC provides a simple log mechanism containing the three classes `Level`, `Logger` and `LogManager`. The `Level` class defines a set of standard logging levels. A `Logger` object is used to log messages for a specific system or application component. `LogManager` is the abstract base class for log managers in ULC. It manages the logger objects for a ULC application. `SimpleLogManager` is the default log manager used for ULC applications. More information about these classes can be found in the [ULC Reference Guide](#).

By default the `SimpleLogManager` is used to log information, both on the client side (UI Engine) and on the server side (ULC). ULC itself logs information at the following levels:

Level SEVERE: Log critical errors and events that stop the session.

Level WARNING: Log errors that do not stop the session.

Level INFO: Log events that occur once for a session.

Level FINE: Log events that occur once for a server round-trip.

Level FINER: Log events that occur once for a request.

Level FINEST: Log additional information such as contents of a request.

To be able to see the information logged by ULC, configure the logging level of the current log manager. See the example provided in Chapter 6.3. By default the level is set to WARNING.

6.2 Configuring the Client-Side (UI Engine) Log Manager

The log level for the UI Engine can be set by providing it as a parameter to the launcher. For a `DefaultAppletLauncher`, the developer can set the `log-level` parameter for the UI Engine applet in the web page:

```
<jsp:plugin
  type="applet"
  code="com.ulcjava.environment.applet.client.DefaultAppletLauncher.class"
  archive="lib/ulc-applet-client.jar,lib/ulc-base-client.jar,
          lib/ulc-servlet-client.jar">
  <jsp:params>
    <jsp:param name="url-string" value="<%= applicationUrl %>" />
    <jsp:param name="keep-alive-interval" value="900" />
    <jsp:param name="log-level" value="WARNING" />
  </jsp:params>
</jsp:plugin>
```

For a *DefaultJNLPLauncher*, specify an argument in the JNLP file:

```
<application-desc
  main-class="com.ulcjava.environment.jnlp.client.DefaultJnlpLauncher">
  <argument>$$context/hello.ulc</argument>
  <argument>900</argument>
  <argument>WARNING</argument>
</application-desc>
```

6.3 Configuring the Server-Side (ULC) Log Manager

A developer can configure the log manager in the application's *start()* method. Select:

- the desired granularity of log information by setting an appropriate level.
- the IO stream where the logger output will be directed.

```
if (LogManager.getLogManager() instanceof SimpleLogManager) {
    SimpleLogManager manager =
        (SimpleLogManager) LogManager.getLogManager();
    manager.setLevel(Level.ALL);
    manager.setLogStream(System.out);
}
```

Alternatively, the log level can be provided as an argument to the container, in which a ULC application is running. For instance, if the ULC application is running as a Servlet then the log level can be specified as a parameter to the Servlet in the `web.xml` file as follows:

```
<servlet>
  <servlet-name>ApplicationApplet</servlet-name>
  <servlet-class>
    com.ulcjava.container.servlet.server.ServletContainerAdapter
  </servlet-class>
  <init-param>
    <param-name>application-class</param-name>
    <param-value>com.ulcjava.sample.hello.HelloApplet</param-value>
  </init-param>
  <init-param>
    <param-name>log-level</param-name>
    <param-value>WARNING</param-value>
  </init-param>
</servlet>
```

Changing the logging configuration in the above described ways affects all ULC sessions running in the same “environment”. E.g., if a ULC application runs as a Servlet *A*, then all ULC sessions associated with *A* have the same log level and the same log stream. However, if there exists a second ULC application, which runs as Servlet *B* in the same web container, then the logging configuration of the ULC sessions of *B* are not affected by changing the logging configuration for *A*. In short, *every Servlet hosting a ULC application offers a separate logging configuration scope* for the ULC sessions

it controls. If a ULC application is deployed as an EJB, then *every instance of that EJB offers a separate logging configuration scope too.*

The different logging configuration scopes only affect the logging that happens *within a ULC session* (including all ULC proxies contained in the session). They do not affect the logging configuration on the environment level, in which the ULC sessions are hosted. E.g., the logging information provided by the Servlet instance that contains a ULC session is outside of these scopes. The same holds for an EJB instance, which contains a ULC session. In other words, logging messages that happen on the servlet execution level or the EJB execution level (outside a ULC session) are not affected by the logging configuration.

The logging configuration on the environment level has a separate, global scope, which can be set by means of a properties file. In the Servlet case, the global scope is the entire web application (usually represented by a `war` file). In the EJB case, the global scope is the entire EJB application (usually represented by an `ear` file).

The properties file to set the global scope is called `ulclog.properties` and maybe be deployed as part of a ULC web application. In the Servlet case, it must be accessible by the ULC Servlet and should be placed under the following path of a corresponding `war` file:

`WEB-INF/classes/ulclog.properties`

In the EJB case, the properties file should be placed in `jar` file, which contains your ULC application. (The `jar` file itself is typically contained in an `ear` file.)

The properties file may have one (optional) property entry:

- `log-level` defines the log level for the resulting logging configuration

If the properties file or its property entry does not exist, then the default log level WARNING is used.

7 Layout Containers

7.1 General Concepts

In many cases, the target machine on which the UI Engine will run is not known at development time. Nevertheless, the layout of a user interface should be visually appealing in different environments. Explicit placement of widgets does not work well in this situation because the exact widget sizes are not known in advance. In addition, the precise horizontal and vertical alignment of widgets and the specification of the resize behavior is a tedious task. ULC provides layout management as an integral part, based on a hierarchical and high-level layout description rather than on the explicit placement of widgets. This layout management handles resize behavior automatically. The layout mechanism of AWT and Swing has the same goals.

ULC supports four layout container types:

- **Basic:** The basic layout containers align widgets in a row/column fashion. These are *ULCBoxPane*, *ULCGridBagLayoutPane*, *ULCGridLayoutPane*, *ULCBorderLayoutPane*, *ULCFlowLayoutPane*, and *ULCBoxLayoutPane*.
- **Stacked:** The stacked layout containers pile page components on top of each other in such a way that only the topmost page is visible. These are *ULCCardPane*, and *ULCTabbedPane*.
- **Special-purpose:** The special-purpose layout containers provide layout management on a higher abstraction level. These are *ULCScrollPane*, *ULCSplitPane*, and *ULCToolBar*.
- **MDI:** MDI layout containers provide layout management by handling a set of internal frames. This is *ULCDesktopPane*.

The widgets *ULCBoxPane* and *ULCGridBagLayoutPane* have the notion of a *cell* and *cell alignment*. A cell is a space which can be empty or can accommodate a single widget. The size of a cell is always equal to or greater than the minimum size of the enclosed widget (if any). If the cell is larger than the preferred size of the widget, the weight and the cell alignment specifies what to do with the extra space. Possible options are to expand the widget until it fills the cell completely or to align the widget within its cell (see Figure 3).

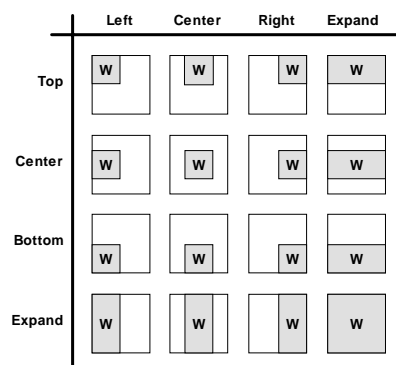


Figure 3: Horizontal and vertical cell alignment

7.2 Using *ULCBoxPane*

The *ULCBoxPane* layout component is based on a few fundamental concepts. Combining these layout strategies recursively enables sophisticated layouts and automatic resize behavior. There are many ways to solve a given layout problem and in general it is not intuitively clear which approach is the least complex and the most appropriate. The following paragraphs outline an overall strategy for designing and implementing a layout using the *ULCBoxPane* layout concepts. In addition, we provide some guidelines and tips for cases with several alternative solutions.

7.2.1 Planning the Layout

Mentally, visualize the layout as a grid. Decide which components will be placed in each grid cell. Decide how that component should size itself when the parent view is resized. For almost every layout a resize strategy has to be defined: which elements should grow if the containing window is enlarged? What happens in a localized version of the layout when a supported language has different string length requirements? Another requirement is aesthetics: how do we keep controls nicely grouped and aligned and not spaced too far apart even if everything may resize drastically?

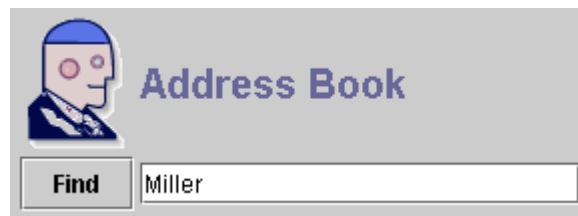


Figure 4: Nested boxes or a two-dimensional box?

Start from Top to Bottom

Identify the top-level elements or groups of elements. Decide whether to use a horizontal, vertical or two-dimensional *ULCBoxPane* layout. Sometimes this is not as simple as it seems. For example: is the top-level layout of Figure 4 a 2x2 box (see Figure 5 left) or is it a vertical box with two rows each containing a horizontal box (see Figure 5 right)?

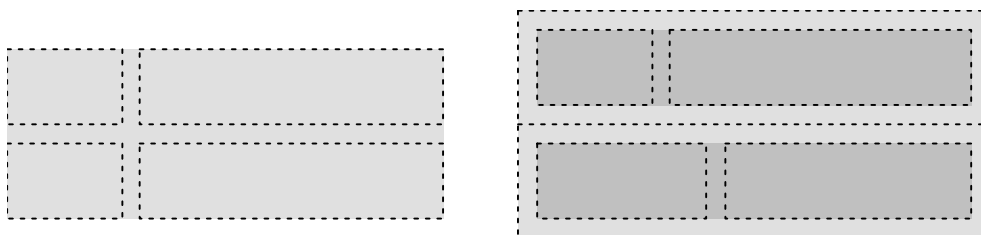


Figure 5: Possible structures for Figure 4

Sometimes an answer can be found by asking whether elements should be kept aligned across horizontal rows. For example, should the label **Address Book** line up with the **Find** text field? If yes, a 2x2 box has to be used. If not (which appears to be more realistic in this case) it is better to nest independent horizontal boxes within one vertical box.

Because constructing box hierarchies is simpler than rearranging them, build and verify the complete box layout before filling in and adjusting the settings of all the non-layout widgets.

Avoid Deep Nesting

Always try to keep your layout structures simple. The most important way to achieve this goal is by avoiding structures nested too deeply. Using two-dimensional boxes instead of nesting boxes simplifies the layout. But this goal sometimes conflicts with a pleasing layout because the two-dimensional box forces elements into a rigid grid and thereby aligns elements, which need not or should not align (see example in Figure 6). Below we describe how spanning makes the use of two-dimensional boxes much more flexible.

Use Spanning

Using spanning it is possible to merge adjacent cells to form larger cells. This is useful if you want to align components with different size requirements within an overall grid. Without spanning, the width and/or height of rows and columns are determined by the widest or tallest element. With spanning, exceptionally large elements can grow into neighboring cells without making their containing rows or columns too wide or tall.

One way to implement the following example (Figure 6) is to nest a 1x3 box (for city, state, and zip) inside the overall 3x2 box. Spanning allows the use of a single overall 3x4 box (with spanning of street and country field) and avoids nesting altogether.

Street:	<input type="text"/>		
City, State, ZIP:	<input type="text"/>	<input type="text"/>	<input type="text"/>
Country:	<input type="text"/>		

Figure 6: Using spanning

Spanning is a very powerful mechanism. You can use a grid layout, which improves the overall aesthetics because elements are aligned and their spacing is more uniform. On the other hand you are not forced to make everything the same size because you can span elements across multiple grid cells. In addition, grid spanning makes it easy to have elements with different sizes to always use multiples of some base cell size, which is visually more appealing than having elements with various unrelated sizes (this concept is called a typographic grid).

When using the *ULCBoxPane*, make sure that for each column, at least one component has its vertical constraint property set to `EXPAND` and for each row, at least one component has its horizontal constraint property set to `EXPAND`. Otherwise, your UI might look odd when extra space is available.

7.2.2 A Simple Address Form Using Nested Boxes

The code to build the form displayed in Figure 6 is shown below:

```
// Create a box with 3 rows and 2 columns
```

```

ULCBoxPane addressBox = new ULCBoxPane(2, 3);

// add the label for Street
addressBox.add(new ULCLabel("Street:"));

// add the field for Street and specify that it must be at
// least 20 columns (characters) wide.
// we also specify the alignment here by choosing the option
// EXPAND_CENTER which defines that this field should expand in the
// horizontal direction when the parent expands to take up all the
// room allocated to it and it should remain vertically
// centered.
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(20));

// add the label for City state zip
addressBox.add(new ULCLabel("City,State,ZIP:"));

// create a Horizontal box to contain the City, State, Zip fields
ULCBoxPane cityStateZipBox = new ULCBoxPane(false);

// add the field for the city and specify that it must be at least 10
// columns wide and expand horizontally when the parent expands.
cityStateZipBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(10));

// add the field for the state and request that it must be at least 2
// columns wide and expand horizontally
cityStateZipBox.add(new ULCTextField(2));

// add the field for the zip and set it to be at least 5 columns wide
// and expand horizontally
cityStateZipBox.add(new ULCTextField(5));

// now add the cityStateZipBox to the parent addressBox and request
// that it expands horizontally
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, cityStateZipBox);

// add the label and field for the Country
// expand horizontally
addressBox.add(new ULCLabel("Country:"));
addressBox.add(ULCBoxPane.BOX_EXPAND_CENTER, new ULCTextField(20));

// Finally we add the addressBox to the parent box/frame and set
// the frame to be visible to display the form. (code not shown)

```

Note that in all *add()* operations of the above example we have not specified the row and column position at which the component should be added. The *ULCBoxPane* widget automatically adds parts starting from the top left corner and adds the widgets in each column until it reaches the last column and then moves to the next row and so on. The order of adding the parts in this case is critical. If for any reason you are unable to build the form in a top-down, left-right order you can use the extended *set()* API of *ULCBoxPane* which takes the row and column within the box at which the component should be added.

The address form demonstrated the use of nested boxes as well as the cell alignment within the box.

7.2.3 A Simple Address Form Using Spanning

Let us expand the above sample to add a **Notes** field that spans two rows and expands when the parent view is expanded.

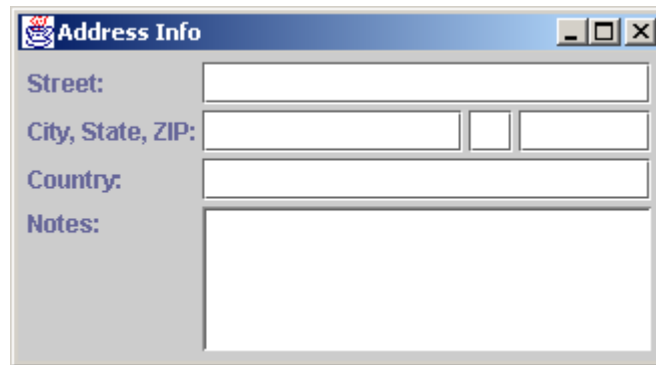


Figure 7: Address form using nested boxes and spanning

The code snippet to add the **Notes** field to the previous *addressBox* is:

```
// add the Notes label
addressBox.add(ULCBoxPane.BOX_LEFT_TOP, new ULCLabel("Notes:"));

// we add a ULCTextArea with 5 rows and a minimum columns size of 10 to
// span vertically over two rows and expand in both directions when
// resized.
addressBox.add(ULCBoxPane.BOX_EXPAND_EXPAND, new ULCTextArea(5, 10));
```

Note that in the previous sample we created a *ULCBoxPane* with only 3 rows but we are spanning the **Notes** field over row 4. This seems wrong and though it is not good practice it works because the *ULCBoxPane* automatically handles additional rows. If we were spanning in the horizontal direction and exceeded the configured column size it would not work since the box would wrap around at the last column and attempt to span to the next row. In fact, experienced ULC developers usually create a box using the following API:

```
// Create a box with n rows and 2 columns
ULCBoxPane addressBox = new ULCBoxPane(2, 0);
```

Since the box will automatically wrap around, we do not need to concern ourselves with the actual number of rows added. We also see that when spanning is used we need to track the current row and current column so that the correct value is fed into the *add* API of *ULCBoxPane*.

8 Drag & Drop

ULC provides support for Drag & Drop operations. A Drag & Drop operation is started with a drag gesture (e.g. click an object and drag the mouse) on the drag source component. The dragged object can subsequently be dropped on a drop target component.

The following ULC components support Drag & Drop operations: *ULCLabel*, *ULCList*, *ULCPasswordField*, *ULCTable*, *ULCTableTree*, *ULCTextArea*, *ULCTextField*, *ULCTree*.

8.1 Drag & Drop Operation Overview

When a Drag & Drop operation is started with an appropriate drag gesture, the ULC component that acts as the drag source creates a drag data object that describes the component-specific items that were selected at the point the drag started. While the selected items are dragged from the drag source towards the drop target (the mouse button is pressed and the mouse is being moved), the drag data object created by the drag source is stored within a *Transferable* instance. In addition, the cursor type of the mouse pointer is continuously being updated depending on the current possible drop target and the actions supported both by the drag source and the drop target.

As soon as the dragged items are being dropped on an ULC component that acts as the drop target, the drop target creates a drop data object that contains a description of the location where the drop occurred. The *Transferable* instance is completed with the drop data object and is sent to the server. On the server-side, the appropriate *TransferHandler* instance that is registered with each ULC component that supports Drag & Drop operations is being notified. First, the *importData()* method of the *TransferHandler* for the drop target ULC component is called to import the data provided by the drag source. During this step, items dragged from the drag source to the drop target can be added to the drop target ULC component. Second, the *exportDone()* method of the *TransferHandler* for the drag source is called to complete the data export. During this step, items moved from the drag source to the drop target can be removed from the drag source ULC component. The actual implementation of the *importData()* and *exportDone()* method depends on the application-specific *TransferHandler* implementation.

8.2 Configuration

Drag & Drop support is configured using the following methods, located on each ULC component that supports Drag & Drop:

Method	Description
<i>setDragEnabled(boolean)</i>	Enables or disables the drag operation support for a specific ULC component. If enabled, a Drag & Drop operation can be started from the ULC component. If not set otherwise, a default implementation of <i>TransferHandler</i> is used to

Method	Description
	control the Drag & Drop operation. If disabled, dragged items from other ULC components can still be dropped onto the ULC component that has drag support disabled.
<i>setTransferHandler(TransferHandler)</i>	Sets the <i>TransferHandler</i> instance that handles data export on drag and data import on drop and thus configures the behavior of the component concerning Drag & Drop operations. For further information on the <i>TransferHandler</i> , see Section 8.4.

ULC provides a default implementation of the *TransferHandler* for each component that supports Drag & Drop. The default implementation is limited to the exchange of plain text data. The operations supported by each component are listed in the following table:

Component	Drag (Data Export)	Drop (Data Import)
<i>ULCLabel</i>	text/plain	-
<i>ULCList</i>	text/plain	-
<i>ULCPasswordField</i>	-	text/plain
<i>ULCTable</i>	text/plain	-
<i>ULCTableTree</i>	text/plain	-
<i>ULCTextArea</i>	text/plain	text/plain
<i>ULCTextField</i>	text/plain	text/plain
<i>ULCTree</i>	text/plain	-

This default behavior also applies to Drag & Drop operations between a specific ULC application and an external application.

Examples

The following example shows how to turn on Drag & Drop support for an *ULCList* component. The default implementation for the *TransferHandler* is used.

```
ULCList list = new ULCList(new String[]{"one", "two", "three"});
list.setDragEnabled(true);
```

8.3 Transfer Data

Data is transferred between the drag source and the drop component using the *Transferable* class. A *Transferable* is a container that contains data objects for specific data.

8.3.1 DataFlavor

A *DataFlavor* instance acts as a key for the associated transfer data. ULC uses the following types of *DataFlavor*:

Data Flavor	Description
DRAG_FLAVOR	DataFlavor that is used to describe the data exported by the ULC component acting as drag source. The data associated with this DataFlavor contains the drag source ULC component and the items selected for dragging.
DROP_FLAVOR	DataFlavor that is used to describe the data exported by the ULC component acting as drop target. The associated data contains the drop target ULC component and information on the location where the item has been dropped.

8.3.2 IDnDData

During a Drag & Drop operation, data exported by the drag source and imported by the drop target is represented as an instance/subclass of *IDnDData*. Both the ULC component that acts as drag source and the component acting as drop target provide an *IDnDData* object. Conceptually, the *IDnDData* objects contains the following information:

- For the drag source, the *IDnDData* object contains information on the drag source component, and on the items that were selected on and dragged from the drag source. The items selected for dragging are described by component-specific data (such as selected indices, rows, columns, tree paths, text positions or text parts) that is contained in the *IDnDData* object created by the drag source ULC component.
- For the drop target, the *IDnDData* contains information on the drop target component and location where the dragged items have been dropped. The drop location is described by component-specific data (such as selected index, row, column, tree path, or text positions) that is contained in the *IDnDData* object created by the drop target ULC component.

ULC uses the following implementations of the *IDnDData* interface:

IDnDData	Component	Structure
DnDLabelData	<i>ULCLabel</i>	<ul style="list-style-type: none">• Component• label text
DnDListData	<i>ULCList</i>	<ul style="list-style-type: none">• component• selected indices
DnDTableData	<i>ULCTable</i>	<ul style="list-style-type: none">• component• selected rows• selected columns
DnDTableTreeData	<i>ULCTableTree</i>	<ul style="list-style-type: none">• component• selected tree paths• selected columns
DnDTextData	<i>ULCTextArea</i> <i>ULCTextField</i> <i>ULCPasswordField</i>	<ul style="list-style-type: none">• component• text position• text
DnDTreeData	<i>ULCTree</i>	<ul style="list-style-type: none">• component• selected tree paths
DnDExternalData	external applications	<ul style="list-style-type: none">• text representation of external data

8.4 TransferHandler

The *TransferHandler* is used to configure the behavior of a specific ULC component during Drag & Drop operations. It is also responsible for handling data import and export. Typically, each ULC component has its own instance of *TransferHandler*.

The *TransferHandler* class defines the following methods:

Method	Description
<i>getSourceActions()</i>	Returns the Drag & Drop actions supported by the ULC component when acting as drag source. Valid actions are: <ul style="list-style-type: none">• <i>TransferHandler.ACTION_NONE</i>• <i>TransferHandler.ACTION_COPY</i>• <i>TransferHandler.ACTION_MOVE</i>• <i>TransferHandler.ACTION_LINK</i> or a combination of them.
<i>getTargetActions()</i>	Returns the Drag & Drop actions supported by the ULC component when acting as drop target. Valid actions are: <ul style="list-style-type: none">• <i>TransferHandler.ACTION_NONE</i>• <i>TransferHandler.ACTION_COPY</i>• <i>TransferHandler.ACTION_MOVE</i>• <i>TransferHandler.ACTION_LINK</i> or a combination of them.
<i>importData(...)</i>	Called by the ULC framework whenever a drop operation on a specific ULC component occurs. This method is used to import dragged data into the ULC component acting as drop target.
<i>exportDone(...)</i>	Called by the ULC framework whenever a drop operation has completed. This callback method is used to modify the ULC component acting as drag source, e.g. for deleting moved items.

Examples

The following example shows how to turn on and configure the Drag & Drop support with a custom *TransferHandler* implementation for an *ULCList*.

```
public class MyTransferHandler extends TransferHandler {
    public int getSourceActions(ULCComponent component) {
        // allow copy and move
        return TransferHandler.ACTION_COPY | TransferHandler.ACTION_MOVE;
    }

    public boolean importData(ULCComponent target, Transferable t) {
        // get transfer data exported by the drag source
        // (here, it must be an ULCList component)
        Object dragData = t.getTransferData(DataFlavor.DRAG_FLAVOR);
        DnDListData listDragData = (DnDListData)dragData;

        // get source list and the indices of dragged list items
        ULCList sourceList = listDragData.getList();
        int[] draggedListIndices = listDragData.getIndices();

        // get transfer data exported by the drop target
```

```

        // (must be an ULCList component)
        Object dropData = t.getTransferData(DataFlavor.DROP_FLAVOR);
        DnDListData listDropData = (DnDListData)dropData;

        // get index of the item at the drop location
        int targetListIndex = listDropData.getIndices()[0];

        // ... handle data import

        // return whether import was successful or not
        return true;
    }

    public void exportDone(ULCComponent src, Transferable t, int action) {
        // check if items have been moved from drag source to
        // drop target and remove items from the drag source list.
        if (action == TransferHandler.ACTION_MOVE) {
            Object dragData = t.getTransferData(DataFlavor.DRAG_FLAVOR);
            DnDListData listDragData = (DnDListData)dragData;
            ULCList sourceList = listDragData.getList();
            int[] movedListIndices = listDragData.getIndices();

            // ... remove items from the list
        }
    }
}

```

9 EJB Development

The EJB architecture defines certain rules that enterprise beans must follow. If a ULC application is deployed into an EJB container, the application is run as a stateful session bean. The ULC session classes and all application classes must therefore adhere to these rules. Note that similar rules may also apply in other containers. As an example a servlet container may support passivation and activation of servlet session by serializing session state.

This chapter shortly discusses these rules. For more information, please refer to the EJB specification and the documentation of your EJB container.

9.1 Serializability

The EJB architecture mandates that beans are serializable, i.e., implement the *java.io.Serializable* interface. EJB containers use this property to passivate the state of beans. The actual implementation of passivation and activation depends on the chosen EJB container.

The ULC framework itself is compliant with this rule. All server-side classes of the ULC modules *base*, *applet*, and *ejb*, and all classes they depend on, are serializable. The full state of a ULC application session along with the state of all widgets can be serialized by the container.

For application classes (including any library classes the application depends on), the application developer must ensure serializability. For convenience reasons, ULC provides all event listener interfaces in a serializable variant as part of the *com.ulcjava.base.application.event.serializable* package, which is useful when implementing anonymous event listener classes.

The following example shows an application that is serializable. Note the statement importing the *com.ulcjava.base.application.event.serializable.IActionListener* instead of the usual *com.ulcjava.base.application.event.IActionListener*.

```
import com.ulcjava.base.application.IApplication;
import com.ulcjava.base.application.ULCButton;
import com.ulcjava.base.application.ULCFrame;
import com.ulcjava.base.application.event.ActionEvent;
import com.ulcjava.base.application.event.serializable.IActionListener;
import com.ulcjava.base.development.DevelopmentRunner;

import java.io.Serializable;

public class Hello implements IApplication, Serializable {
    private ULCFrame fFrame = null;
    private ULCButton fButton = null;

    public void start() {
        fFrame = new ULCFrame("Hello world");

        fButton = new ULCButton("Some action");
        fButton.addActionListener(new IActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.out.println("Some action");
            }
        });
    }
}
```

```
        }
    });

    fFrame.add(fButton);
    fFrame.setVisible(true);
}

public void activate() {
    // release any unneeded resources and transient data here
}

public void passivate() {
    // reopen resources and restore transient data here
}

public void stop() {
}

public void handleMessage(String message) {
}
}
```

9.2 Programming Restrictions

The EJB specification defines a range of programming restrictions on an enterprise bean and all classes it depends on. These restrictions are required to ensure that enterprise beans are portable and can be deployed in any compliant EJB container. As an example, code that is deployed in an EJB container must not attempt to manage (e.g., start) threads and is not allowed to use read/write static variables.

All server-side classes of the ULC modules *base*, *applet*, and *ejb* are compliant with the restrictions as defined by EJB specifications 1.1 and 2.0 (see [22]).

The application developer must ensure that all application classes (including any library classes the application depends on) also follow these restrictions.

10 Using the Pause / Resume API

ULC provides functionality to allow the pause and resume of a running application instance including its editing state, e.g. current windows opened, table scroll positions. This means you can pause the application. This will terminate the client side engine, while the server-side application part is not terminated. Once a client requests the server to resume an application, it can be downloaded to the client again, showing the UI in the same state as when the application was paused.

10.1 Example showing Pause and Resume Functionality

The following example shows how to use the pause and resume functionality in such a way, that a single application instance can be started, paused, resumed and stopped. This example shows pause and resume with a custom *TeamMembersResumeLauncher* that runs in the development environment.

10.1.1 Launcher Implementation

To support pause and resume custom launcher and container adapter classes have to be implemented. This example is based on the *TeamMembers* sample application and is intended for the development environment. Because of this, we will not implement a container adapter for this example.

The *TeamMembersResumeLauncher* creates a small Swing-based application, that offers buttons to start, pause, resume and stop the application. Depending on the state of the launcher, the buttons are enabled or disabled. The *TeamMembersResumeLauncher* implements custom session state listeners to start and resume the application.

```
...
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            TeamMembersResumeLauncher resumeLauncher =
                new TeamMembersResumeLauncher(TeamMembers.class);
            resumeLauncher.start();
        }
    });
}

public void start() {
    if (ClientEnvironmentAdapter.getBrowserService() == null) {
        ClientEnvironmentAdapter.setBrowserService(
            new AllPermissionsBrowserService());
    }
    if (ClientEnvironmentAdapter.getFileService() == null) {
        ClientEnvironmentAdapter.setFileService(
            new AllPermissionsFileService());
    }

    final JButton startButton = new JButton("Start");
    startButton.setEnabled(true);
}
...
```

The action listener installed for the start button creates a *DevelopmentConnector* and a *UISession*. After installing a *ISessionStateListener* the session gets started. In the *ISessionStateListener* the connector gets started and the *startApplication* command is sent. Note that the *sessionEnded* method of the connector is not stopped.

```
public void actionPerformed(ActionEvent e) {
    fConnector =
        new DevelopmentConnector(fApplicationClass.getName(),
            new Properties(), ConnectionType.UNLIMITED);
    fSession = new UISession(fConnector, new Properties());
    fSession.addSessionStateListener(
        new ISessionStateListener() {
            public void sessionStarted(UISession session) throws Exception {
                session.startConnector();
                session.sendStartApplication();
            }

            public void sessionEnded(UISession session) throws Exception {
            }
        }
    );
    ...
}
```

The action listener installed for the resume button creates and starts a new client session. Next, in the *ISessionStateListener* the *resumeApplication* command is sent to the server side. Note that in this development environment example the connector constructed in the start button action listener is used again for the new client session and this connector still has a connection to the container adapter holding the instance of the server side session. Finally the session is started.

```
public void actionPerformed(ActionEvent e) {
    new Thread() {
        public void run() {
            fSession = new UISession(fConnector, new Properties());
            fSession.addSessionStateListener(new ISessionStateListener() {
                public void sessionStarted(UISession session)
                    throws Exception {
                    session.sendResumeApplication();
                }
            });
            ...
            fSession.start();
        }
    }.start();
    ...
}
```

In the pause button's action listener a *pauseApplication* command is sent to the server side.

```
public void actionPerformed(ActionEvent e) {
    new Thread() {
        public void run() {
            fSession.sendPauseApplicationAndWait();
        }
    }.start();

    ...
}
```

In the stop button's action listener a *stopApplication* command is sent to the server side and finally the connector is stopped.

```
public void actionPerformed(ActionEvent event) {
    new Thread() {
        public void run() {
            fSession.sendStopApplicationAndWait();
            try {
                fSession.stopConnector();
            } catch (ConnectorException e) {
                e.printStackTrace();
            }
        }
    }.start();

    ...
}
```

10.1.2 Server-Side Implementation

For this example no server-side implementation is necessary. In the development setup, the *DevelopmentConnector* creates a *DevelopmentContainerAdapter* that will have the *ULCSession* to process the requests. The session has support for pause and resume built in.

In a more advanced scenario a list of paused sessions could be presented on the client, which would require a corresponding container adapter to be implemented.

11 Best Practices

This section describes best practices that might be helpful to ULC application developers. In addition, it points out some issues to avoid when developing with ULC.

Use Callback Functions

Notification is always done through callback functions, which are realized as listener interfaces. It is not possible to stop somewhere in your server-side code and wait until the user triggers something and to only then continue the server-side code on the next line. Your code got triggered as a reaction to a user request and we have to return a response to finish the round-trip. This is different from a pure Swing application.

Avoid Using Static Variables

Avoid the use of static variables because these are shared by all application sessions. The session that changes the variable wins. In most cases, this is not desired.

Use the *ApplicationContext.setAttribute() / getAttribute()* API to keep the session global state that you can access statically.

The only exception from this rule are constants. However, constant references to mutable objects should be avoided as well.

Avoid Sharing Proxies Between Sessions

ULC uses a unique identifier (*object id*) to address each proxy on the client and on the server side, respectively. Since this identifier is only unique within the current session, sharing components across different sessions is not allowed. All framework classes with the prefix “*ULC*” are subclasses of proxies. Do not use *ULCComponents* in more than one place.

The same instance of a *ULCComponent* cannot be added to different panes at the same time. This behaviour is analogous to Swing. Instances of subclasses of *ULCComponent* must not be used in more than one place. A *ULCComponent* must not be put in more than one container.

This rule holds only for *ULCComponents*, all other proxies can be used in more than one place, e.g. icons, enablers, data types.

Share Objects Whenever Possible

Share objects to keep the server session size low, e.g. models, icons, borders.

A large server session size will cause problems in a clustered environment when the session has to be exchanged between nodes of the cluster.

Use Uniform Renderer Components

Currently, you cannot modify the text in the renderer of any widget. Therefore, make sure your widget's model returns the text to be displayed. As an alternative, provide a decorator model that converts your data to its UI representation.

ULC frequently invokes the callback methods of renderers. Therefore you should be very careful to implement efficient callbacks, e.g. it is better to configure the renderer components with existing objects rather than with new objects.

Try to use uniform renderer components, e.g. avoid color gradients in tables. Non-uniform renderer components result in a lot of traffic as ULC cannot optimize communication in such a case.

Avoid Asynchronous Event Handling

Do not use asynchronous event handling, except when you are aware of its consequences.

Thread-Safety

Do not access ULC components except from within the ULC thread. Otherwise, the behaviour of your application is not specified. See the *Polling Timer* contribution on the [ULC Code Community website](#).

Serializable Session

Keep your session serializable in order to allow proper integration into Servlet and EJB application servers. This means that all objects having your *IApplication* instance as an ancestor should be serializable and any objects put in the *ApplicationContext* should be serializable as well. See the *Serializable Check* contribution on the [ULC Code Community website](#).

Session Termination

Always terminate a session by calling *ApplicationContext.terminate()*. Never use *System.exit()*.

Reuse Components

Keep the server and client memory footprint low by reusing ULC components like *ULCFrame*, *ULCWindow*. Hold these components in a pool and reconfigure them before display. Keep your pool in the *ApplicationContext*.

Simulate Low Bandwidth

Discover client-server traffic and communication bottlenecks already during development time. This can be achieved by launching ULC's *DevelopmentRunner* using the *-useGui* program parameter and changing the communication speed to a lower rate.

Use the GUI Provided with DevelopmentRunner

Watch the number of server roundtrips for a user operation. You can reduce this by bunching requests if your GUI allows you to compromise on responsiveness. For instance, by sending all the data from all the fields in a form.

Watch the sizes of the requests. Use this data to optimize the data transfer between ULC application and the UI Engine.

Simulate low bandwidth to discover client-server traffic and communication bottlenecks already during development time.

Check Reachability Through Firewalls and Proxies

Make sure your application can be accessed from remote clients. If you fail to access your application, this might be due to firewall or proxy settings on the client side or server side. Use the *Ping* tool available on the [ULC Code Community website](#) to narrow down the problem.

References

- [1] Jakarta Tomcat servlet container
<http://jakarta.apache.org/tomcat/>
- [2] Java Web Start
<http://java.sun.com/products/javawebstart/>
- [3] J2EE tutorial on web application archives
<http://java.sun.com/j2ee/tutorial/doc/WCC3.html>
- [4] Eclipse
<http://www.eclipse.org/>
- [5] Jakarta ORO regular expression package
<http://jakarta.apache.org/oro/>
- [6] HTML 4.0.1 specification
<http://www.w3.org/TR/html4/>
- [7] Java Secure Socket Extension (JSSE) 1.0.3
<http://java.sun.com/products/jsse/index-103.html>
- [8] Java Security Architecture
<http://java.sun.com/j2se/1.3/docs/guide/security/index.html>
<http://java.sun.com/j2se/1.4.1/docs/guide/security/index.html>
- [9] JNLP forum thread describing the protocol handler workaround
<http://forum.java.sun.com/thread.jsp?forum=38&thread=71335>
- [10] Java Plug-In tags
<http://java.sun.com/products/plugin/1.3/docs/tags.html>
<http://java.sun.com/products/plugin/versions.html>
- [11] Using the conventional APPLET tag with Java Plug-In 1.3.1 and 1.4
http://java.sun.com/products/plugin/1.3.1_01a/faq.html
http://java.sun.com/j2se/1.4/docs/guide/plugin/developer_guide/applet_tag.html
- [12] "usePolicy" Permission
<http://java.sun.com/products/plugin/1.3/docs/netscape.html#use>
- [13] LiveConnect
http://developer.netscape.com/docs/technote/javascript/liveconnect/liveconnect_rh.html
- [14] Calling an applet from Java Script
<http://java.sun.com/products/plugin/1.3/docs/jsobject.html>.
- [15] InstallAnywhere
<http://www.installanywhere.com/>
- [16] InstallShield
<http://www.installshield.com/>
- [17] Initializing an initial context with applet parameters
<http://java.sun.com/j2se/1.3/docs/api/javax/naming/Context.html#APPLET>
- [18] Canoo Engineering AG contact address
ulc-info@canoo.com

-
- [19] Introduction to servlet technology
<http://java.sun.com/products/servlet/index.html>
 - [20] Servlet specifications
<http://java.sun.com/products/servlet/download.html>
 - [21] Introduction to EJB technology
<http://java.sun.com/products/ejb/index.html>
 - [22] EJB specifications
<http://java.sun.com/products/ejb/docs.html>
 - [23] J2EE Tutorial
<http://java.sun.com/j2ee/tutorial/index.html>
 - [24] Packaging J2EE applications
<http://java.sun.com/j2ee/tutorial/doc/Overview4.html>