



macromedia®  
**CENTRAL™**

Developing Central Applications

## Trademarks

1 Step RoboPDF, ActiveEdit, ActiveTest, Authorware, Blue Sky Software, Blue Sky, Breeze, Breezo, Captivate, Central, ColdFusion, Contribute, Database Explorer, Director, Dreamweaver, Fireworks, Flash, FlashCast, FlashHelp, Flash Lite, FlashPaper, Flex, Flex Builder, Fontographer, FreeHand, Generator, HomeSite, JRun, MacRecorder, Macromedia, MXML, RoboEngine, RoboHelp, RoboInfo, RoboPDF, Roundtrip, Roundtrip HTML, Shockwave, SoundEdit, Studio MX, UltraDev, and WebHelp are either registered trademarks or trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

## Third-Party Information

This guide contains links to third-party websites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

**Copyright © 1997-2005 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without written approval from Macromedia, Inc. Notwithstanding the foregoing, the owner or authorized user of a valid copy of the software with which this manual was provided may print out one copy of this manual from an electronic version of this manual for the sole purpose of such owner or authorized user learning to use such software, provided that no part of this manual may be printed out, reproduced, distributed, resold, or transmitted for any other purposes, including, without limitation, commercial purposes, such as selling copies of this documentation or providing paid-for support services.**

## Acknowledgments

Project Management: JuLee Burdekin

Writing: Jay Armstrong, Jody Bleyle, Alec Flett, David Jacowitz, Shimul Rahim

Editing Management: Rosana Francescato

Editing: Linda Adler, Mary Ferguson, Noreen Maher, Barbara Milligan, Antonio Padiál, Lisa Stanziano

Production Management: Patrice O'Neill

Media Design and Production: Adam Barnett, Christopher Basmajian, Aaron Begley, John Francis

Special thanks to: Mike Chambers, Phillip Kerman, Kevin Lynch, Randy Nielsen, Vijay Shah

Third Edition: January 2005

Macromedia, Inc.  
600 Townsend St.  
San Francisco, CA 94103

# CONTENTS

<b>INTRODUCTION:</b> About This Guide . . . . .	7
Intended audience . . . . .	8
Navigating the documentation map . . . . .	9
Additional resources . . . . .	9
Typographical conventions . . . . .	9
 <b>CHAPTER 1:</b> Getting Started . . . . .	11
System requirements . . . . .	11
Installing Macromedia Central . . . . .	11
Installing the Software Development Kit (SDK) and components . . . . .	12
The FirstApp application . . . . .	13
Adding the final touches . . . . .	18
Taking the next steps . . . . .	18
 <b>CHAPTER 2:</b> Understanding the Macromedia Central Environment . . . . .	19
About the Macromedia Central framework . . . . .	19
Elements of the Central environment . . . . .	20
Elements of a Central application . . . . .	22
The Macromedia Central product user experience . . . . .	24
Central development workflow . . . . .	25
Programmatic flow of a typical product . . . . .	28
Typical data storage and access techniques . . . . .	28
 <b>CHAPTER 3:</b> Building a Central Application . . . . .	33
Macromedia Central application development workflow . . . . .	33
Migrating from version 1.0 . . . . .	34
Initializing an application . . . . .	36
Implementing the application methods . . . . .	37
Implementing mx.central.Application . . . . .	38
Using the shell API in an application . . . . .	44
Passing data among product parts . . . . .	47
Working with preferences . . . . .	57
Tracking network status . . . . .	64
Caching data locally . . . . .	64
Using web services . . . . .	68

Using regular expressions. . . . .	75
Providing custom context menus. . . . .	75
Using the Blast feature to share data across applications. . . . .	76
Accessing information across domains. . . . .	77
<b>CHAPTER 4: Creating Pods . . . . .</b>	<b>81</b>
Creating a pod. . . . .	81
Controlling pods. . . . .	86
Implementing the pod API . . . . .	88
Communicating between a pod and the Console. . . . .	90
<b>CHAPTER 5: Creating an Agent . . . . .</b>	<b>93</b>
Designing an agent . . . . .	93
Creating an agent SWF file . . . . .	94
Starting an agent . . . . .	95
Stopping an agent . . . . .	96
Determining the status of an agent . . . . .	97
Implementing the agent API . . . . .	97
<b>CHAPTER 6: Creating Notices . . . . .</b>	<b>101</b>
Creating a notice . . . . .	101
Responding to notices . . . . .	103
Guidelines for using notices. . . . .	104
<b>CHAPTER 7: Using the Blast Feature . . . . .</b>	<b>105</b>
Sending data from an application . . . . .	106
Receiving data . . . . .	108
Sending data from pods. . . . .	108
Registering supported data types in the product.xml file . . . . .	109
Defining your own data type schema. . . . .	110
Choosing a schema format . . . . .	110
Defining your own data type schema. . . . .	113
Selected item storage . . . . .	113
Data type reference . . . . .	114
<b>CHAPTER 8: Designing for Central Best Practices . . . . .</b>	<b>121</b>
Configuring Macromedia Flash. . . . .	121
Application user interface . . . . .	122
Central coding conventions. . . . .	122
Optimizing SWF files . . . . .	125
Testing an application . . . . .	125
Converting existing Flash applications into Central applications . . . . .	126
<b>CHAPTER 9: Deploying Central Applications . . . . .</b>	<b>127</b>
Deploying an application . . . . .	127

<b>CHAPTER 10: API Reference</b> .....	133
Central API .....	133
Flash API Deltas .....	136
Agent object .....	137
AgentManager object .....	144
Application object .....	171
Central object .....	184
Console object .....	186
DataProviderClass object .....	218
FileReference object .....	238
FileReferenceList object .....	268
LCDDataProvider object .....	271
LCService object .....	295
Log object .....	300
MD5 object .....	303
MovieClip object .....	305
PendingCall object .....	306
Pod object .....	314
RegExp object .....	325
RPC object .....	331
RPCFactory object .....	334
SelectedItem object .....	335
Shell object .....	338
SOAPCall object .....	381
String object .....	383
WebService object .....	384
XML object .....	392
 <b>CHAPTER 11: The product.xml File</b> .....	395
Sample product.xml file .....	395
Product.XML schema .....	396
Detailed product.xml example .....	410
 <b>INDEX</b> .....	413



# INTRODUCTION

## About This Guide

This guide describes the Macromedia Central framework and explains how to create products for deployment in Macromedia Central.

Macromedia Central is a centralized environment optimized for running Internet applications. The Central platform provides an API for seamless communication and data sharing among applications, as well as a distribution mechanism that takes advantage of the ubiquity of Macromedia Flash Player. Central also provides an API for caching data and other assets, thus allowing data-driven applications to be used regardless of whether they are online or offline.

Macromedia Central has a number of benefits for both end users and developers.

### **Benefits for the end user**

Macromedia Central is a lightweight environment serving up applications that provide a richer interface to the Internet than is possible with applications hosted within a browser. The Central environment provides a number of key features, including the following:

**Consistent user experience** The Macromedia Central SDK provides a core set of UI components that help enforce consistent functionality and look among applications. This makes it easier for end users to learn and move between applications.

**Online/offline support** Central provides an API for developers that makes it easy to cache data and other assets for offline use. This allows users to use their applications and access their data regardless of whether they are online or offline.

**Cooperative applications** Central provides a unified environment where all applications can work with each other and share information.

**Dynamic notification** Users can set up notices on a per-application basis, and thus are notified when specific events occur. For example, a stock application may allow a user to set up a notice when a stock reaches a certain price.

## Benefits for the developer

Based on Macromedia Flash Player and the ActionScript scripting language, Macromedia Central provides an environment and API for taking advantage of skills that use Flash to create and distribute applications.

**Ease of installation** Macromedia Flash Player 6 (and later versions) has built-in support for installing Central applications as well as the Central environment. This allows developers to take advantage of the ubiquity of Flash Player to reach virtually everyone on the Internet.

**Central environment** Central provides an easy-to-use, consistent UI for deploying and running applications.

**Central components** The Central SDK provides a core set of UI components that allow developers to quickly create applications and provide consistent user interfaces for users.

**Central application programming interface (API)** In addition to providing full support for Macromedia Flash 6 ActionScript and APIs, Central provides a set of APIs that simplify application development and management.

**Web services support** Central has a full-featured web services API that allows developers to quickly create applications that interface with data and web services on the Internet.

**Regular expression support** Central has a complete regular expression API to simplify the manipulation of text and strings.

**Data caching API** Central provides a number of APIs for storing data and assets locally for both online and offline use.

**Dynamic image loading** Central provides support for dynamically loading JPEG (including Progressive JPEG) and GIF files.

**Auto update support** Central provides support for easily and automatically pushing out application updates to users.

## Intended audience

This book is intended for Flash developers with a good working knowledge of Flash and ActionScript, as well as for developers of other interactive and enterprise applications. This book assumes that the reader has access to the Flash MX documentation and Macromedia Developer Center resources at [www.macromedia.com/devnet](http://www.macromedia.com/devnet).

**Note:** The format of [Chapter 10, “API Reference,” on page 133](#) is based on ActionScript 2.0, which is a feature of Macromedia Flash MX 2004. The primary difference between the two versions of ActionScript is that ActionScript 2.0 uses classes as the main unit of functionality, while ActionScript 1.0 uses objects as the main unit of functionality. Users of Flash MX and ActionScript 1.0 should simply substitute the word *object* for the word *class* in the reference chapter.



## Navigating the documentation map

This book explains the details of developing applications for Macromedia Central. It assumes that the reader has a general knowledge of Macromedia Flash. Documentation about Flash and related products is available separately.

- For a complete description of the Macromedia Central user interface, see *Using Macromedia Central*, included in the SDK.
- For information about Macromedia Flash, see *Using Flash* and *Flash ActionScript Language Reference*, both available from the Help menu within the Flash authoring environment.
- For information about using components in Central applications, see *Building Central Applications with Components*, included in the SDK.

## Additional resources

The Central Developer Center at [www.macromedia.com/go/central\\_dev\\_center](http://www.macromedia.com/go/central_dev_center) is updated regularly with the latest information on Central. It also contains articles, tutorials, samples, and other resources to help developers get the most out of the Central environment.

The Macromedia Central Forums provide developer-to-developer interaction to discuss application development for Macromedia Central. The forums can be found at <http://webforums.macromedia.com/central/>.

The Macromedia Flash Support Center website at [www.macromedia.com/go/flash\\_support](http://www.macromedia.com/go/flash_support) is updated regularly with the latest information on Flash. It also contains advice from expert users, advanced topics, examples, tips, and other updates. Check the website often for the latest news on Flash and for tips on getting the most out of the program.

## Typographical conventions

The following typographical conventions are used in this book:

- `Code font` indicates ActionScript statements, XML tag and attribute names, and literal text used in examples.
- *Italic font* indicates placeholder elements in paths. For example, */settings/myPrinter/* means that you should specify your own location for *myPrinter*.
- *Code italic* indicates a placeholder element, such as an ActionScript parameter or object name, that you replace with your own text when writing a script.
- **Bold font** indicates a verbatim entry.



# CHAPTER 1

## Getting Started

This chapter explains how to set up and install Macromedia Central to develop applications. This document includes a step-by-step tutorial for creating a basic application you can use to ensure that Central is installed correctly and to walk you through the basic application development process. You can also use this application as a basis for creating more complex applications.

To learn more about the Central interface, read [Chapter 2, “Understanding the Macromedia Central Environment,” on page 19](#). You can return to this chapter when you are ready to start building your first Central application.

### System requirements

To develop applications for Macromedia Central you must have a tool (such as Macromedia Flash MX 2004) that can generate SWF files.

To deploy Central applications, you must have access to a web server.

To run Macromedia Central, you must run Central on a computer that meets the Central system requirements. You can find these requirements at [www.macromedia.com/go/central\\_sysreq](http://www.macromedia.com/go/central_sysreq).

### Installing Macromedia Central

Macromedia Flash Player 6 (6.0.65.0) and later has built-in support for installing Macromedia Central. If a user attempts to install a Central application on a computer where the Central environment is not already installed, Flash Player prompts the user to install Central first. Then, after Central is installed, Central also installs the application originally requested.

If a user attempts to install a Central application on a computer where the Central environment is already installed, Flash Player installs the application into Central.

**To download Central from the Macromedia website:**

1. Make sure you have the latest version of Macromedia Flash Player installed on your system. (You'll need Flash Player 6 (6.0.65.0) or later.)
  - You can find out which Flash Player version is installed at the Macromedia Flash Player Download Center at [www.macromedia.com/support/flash/ts/documents/test\\_version.htm](http://www.macromedia.com/support/flash/ts/documents/test_version.htm).
  - You can install the latest version of Macromedia Flash Player from the Macromedia Flash Player Installation page at [www.macromedia.com/go/getflashplayer](http://www.macromedia.com/go/getflashplayer).
2. To install Central, go to the Central Installation page at [www.macromedia.com/go/install\\_central](http://www.macromedia.com/go/install_central) and follow the instructions.

## Installing the Software Development Kit (SDK) and components

The Central Software Development Kit (SDK) ZIP file contains documentation, sample files, utility files, components and other resources. These can be used to design and develop applications for Central.

**To install the Central SDK:**

- Extract the contents of the ZIP file to a directory of your choice. You will see the following directory structure under the root directory you chose:

File/Directory name	Description
AuthoringExtensions	Directory containing the Central components, Central Debug Panel, Central Publishing Tool and the intrinsic class files.
Documentation	Directory containing all of the SDK help files, including the help files for the components, and the example files referred to in the documentation.
Samples	Directory containing sample files that the Central team has put together for you to explore.
Utilities	Directory containing utilities that make Central development and debugging easier.
ReleaseNotes.html	File containing the latest information regarding the product, including known issues.
StartHere.html	File listing all of the SDK parts.

**To install the authoring extensions:**

1. Make sure you have the Macromedia Extension Manager installed. You can download the Extension Manager from [www.macromedia.com/exchange/em\\_download](http://www.macromedia.com/exchange/em_download).
2. Locate the AuthoringExtensions.mxp file in the AuthoringExtensions subdirectory under the SDK directory.
3. Open the AuthoringIntegration.mxp file in the Macromedia Extension Manager to install the components automatically in the Macromedia Flash authoring tool.

**To view the Central components:**

1. Start the Macromedia Flash authoring tool.
2. Select Window > Components.  
The Components panel appears.
3. Open Central Components from the Components panel.

**To view the Central Product Setup Wizard tool:**

1. Start the Macromedia Flash authoring tool.
2. Select Commands > Central Product Setup.  
The publishing tool appears.

**To use the Central Debug panel:**

1. Start the Macromedia Flash authoring tool.
2. Select Window > Other Panels > Central Debug Panel.  
The Central Debug Panel panel appears.
3. Start Macromedia Central.

**To install the sample files provided with the SDK:**

1. For each sample, copy the entire subdirectory named *sampleapplication*Installer (for example, StockWatcherInstaller) to a web server directory (local or remote web server).
2. Use your web browser to navigate to that directory, and open the page installer.html.
3. Click the installation badge to install application.

## The FirstApp application

You can find a sample application named FirstApp in the Documentation/pdf/Dev\_Guide\_Examples directory of the SDK. This is a very simple application that dynamically displays text after the application is loaded into Central. It should provide you with a basic idea of the process for creating, testing and installing applications for Central.

### FirstApp files

The complete source files for FirstApp can be found in the FirstApp directory, which is a subdirectory in the Documentation/pdf/DevGuide\_Examples directory. It contains the following files:

Filename	Description
FirstApp fla	Macromedia Flash authoring file that contains the FirstApp design and ActionScript.

Filename	Description
product.xml	Product descriptor XML file that tells Central about the application. It is used at installation time to describe the application to the user and to download all the required pieces. The tag library for this file can be found in <a href="#">Chapter 11, “The product.xml File,” on page 395</a> .
icons/40x40.swf	The SWF file for the FirstApp icon. (To use the icon for your application, replace the SWF file with your own file.)

## Re-creating the FirstApp application

**Note:** The FirstApp created here is a simpler version of the FirstApp sample provided with the Beta SDK. That sample contains more calls and uses a pod.

Re-creating the FirstApp application can help you understand the basics of Macromedia Central application development. The following procedures explain how to create the most basic application that can be successfully downloaded and installed into Central.

Development consists of the following five steps:

1. [Creating a Macromedia Central application](#) within the Flash authoring environment.
2. Obtaining a product ID for the application from [www.macromedia.com/go/central\\_productid](http://www.macromedia.com/go/central_productid).
3. [Preparing the product.xml file](#).
4. Installing the application into Central.
5. Testing and debugging the application within Central.

## Creating a Macromedia Central application

Follow the steps below to re-create the FirstApp application.

### To create the FirstApp application:

1. In the Macromedia Flash authoring tool, create a new file and name it FirstApp fla. In the Publish Settings tab, make sure that the file is set to publish to Flash Player 6 (File > Publish Settings > Flash).
2. Create three new layers on the Timeline named *Actions*, *Title*, and *TextField*. The layer names are not important, but they help organize the source file.
3. On the Title layer, create a static text field and add the following text: **My First Application**.
4. Switch to the TextField layer and create a dynamic text field with an instance name of *fFirstText*. This will be used to dynamically display a message when the application is successfully loaded into Central.
5. Select the first frame of the Actions layer. In the Actions panel, type the following code (you can omit the comments):

```
// called by Central when application is loaded
onActivate = function(shell, appId, shellId, baseTabIndex, appData)
{
    // Display text in text field.
    fFirstText.text = "Application has loaded!"
}
```

```

        // Set the message within the Central status bar.
        shell.setStatus("Application has loaded");
    }

    // called by Central when application is unloaded. Clean up any global
    // resources, and close any local connection and socket connections here.
    onDeactivate = function(shell, appId, shellId, baseTabIndex, appData)
    {
    }

    // lets the Central shell know the app is loaded.
    // The shell will then call the onActivate() method.
    mx.central.Central.initApplication(this, this);

```

6. Save the FLA file.

7. Publish the FLA file (File > Publish). This creates a SWF file that contains your application.

Before you move on, take a minute to examine the ActionScript code in the application. The following functions are the bare minimum that all Central applications should contain.

- `onActivate` Callback function called by Central when the application has loaded. It is passed parameters with information about and references to the Central environment.
- `onDeactivate` Callback function called by Central when the application is about to be closed. You should include code here to clean up any resources that may have been used by your application, such as global variables, socket connections, or shared objects.
- `mx.central.Central.initApplication` Function call that tells the Central environment that your application is ready to start running; `onActivate` is called after this method is called.

**Note:** You can find a more detailed description of these and other functions in [Chapter 10, “API Reference,” on page 133](#).

The `Central.initApplication` method is called as soon as your application is loaded into Central. It is passed two parameters, both of which are references to your application. Central uses these parameters to communicate back to your application.

After `Central.initApplication` is called, Central calls the `onActivate` callback function for the application. This is where your application should begin to initialize itself. The `onActivate` function in the `FirstApp` application does only two things. First, it sets the text in the dynamic text field to specify that the application has loaded and initialized. Second, it uses a reference to the Central shell passed into the `onActivate` method, to place a message on the Central status bar.

The `onDeactivate` function is called when the application is about to be closed, and should be used to clean up any resources, such as global variables, used by the application. In the example, no resources were used and therefore no code is needed here. However, consider including the function, even if you do not need it initially, because it can help remind you to add cleanup code when necessary.

## Obtaining a product ID

All applications installed into Macromedia Central must have a product ID (the product ID is included within the product.xml file, see “[Preparing the product.xml file](#)” on page 16). This includes applications that you install into Central for development and testing purposes. The StartHere.html page of the SDK provides a product ID number you can use for testing.

To obtain a product ID, go to [www.macromedia.com/go/central\\_productid](http://www.macromedia.com/go/central_productid) and follow the instructions.

## Preparing the product.xml file

The product.xml file provides Central with the information necessary to install, run, and manage your application. Central loads and reads this file when your application is installed by a user. In order for Central to be able to use the product.xml file, it must be in the same domain as all other files that will be installed as part of your application.

Follow these steps to prepare a customized version of the product.xml file for your application.

### To customize a product.xml file:

1. Create a copy of the product.xml file from the FirstApp sample application.
2. Open your copy of the product.xml file and change the name attribute for the product and application tags to the name of your application—in this case FirstApp.
3. Change the application src path to point to the location of your SWF file. This path can be relative to the location of the product.xml file. All application files must be located in the Internet domain as the product.xml file.
4. Reference an icon. You can create your own icon and specify its location in the two icon src tags of the product.xml file. The icon files should be a 35 x 35 pixel SWF file for the standard toolbar, and a 23 x 23 pixel SWF file for the small toolbar. Central will scale larger icons to fit the toolbar states. If you prefer, simply reference the icon file that comes with the sample application. If no icon is specified, a default icon is used.

## Using the installation badge to install your application

To install your application into Central, you must have a Flash-based installer that contains the code to prompt Flash Player to install your application. The SDK includes a standard installation badge that you can use to install your application.

The installation badge loads the data from the product.xml file and uses that to install the application.

### To use the installation badge:

1. Copy the installation.swf and installation.html files from the Utilities/InstallationBadge directory on the SDK.
2. Place the files into the same directory as your product.xml file.
3. Copy the directory to a web server (local or remote).
4. Load the installation.html page in a web browser.
5. Click the installation badge to begin the installation of your application.



The badge detects whether the user has the minimum version of Flash Player necessary to install Central and Central applications. If they do not, it will direct the user to install a newer Flash Player version.

The installation badge loads the application name, description and icon specified within the product.xml file.

## Testing and debugging your application within Central

Because applications running within Central must have access to the Central API, they must be tested and debugged within the Central environment. This means that the development process differs slightly from traditional Flash development.

There are two main steps to testing and debugging within the Central environment. The first is publishing a new SWF file directly into Central. This is possible once you have installed your application into Central. The second is viewing runtime debug and trace information from the application.

### To publish directly into the Central environment:

1. Find the location where your application files were installed by Central. In Windows, this will be similar to: C:\Documents and Settings\USERNAME\Application Data\Macromedia\Central\#Central\<random directory>\DOMAINNAME\<subdirectory>. On the Macintosh, this will be similar to: Hard Drive/Users/USERNAME/Library/Application Support/Macromedia/Central/#Central/<random directory>/DOMAINNAME/APPNAME/.
2. Open your application within the Flash authoring environment.
3. Open the Publish Settings dialog box (File > Publish Settings).
4. Set the file to publish into the directory located above. Make sure that the file is published with the same filename that was installed by Central.
5. Click OK to save the settings.

To test your application, you must first publish it (File > Publish). This compiles the SWF file into its application directory within Central. You can then view the updated application file by switching to Central and reloading the application by switching to another application within Central and then switching back. If you are testing a pod, you must close the pod and reopen it. If you are testing an agent, you must restart Central.

After the application is running within the Central environment, you can use the Central Debug panel to view runtime debug information from the application. The Central Debug panel runs within the Flash authoring environment and can be installed from the SDK.

### To test your application using the Debug panel:

1. Open the Debug panel in the Flash authoring environment. In Flash MX, select Windows > Central Debug Panel. In Flash MX 2004, select Windows > Other Panels > Central Debug Panel.
2. Make sure that the Omit Trace Actions option is deselected in the public settings (File > Publish Settings > Flash).
3. Add trace statements to your application. You can pass in various data types including strings, objects and arrays.
4. Test your application within Central. Any trace statements encountered during the application's runtime will be printed out into the Debug panel within the Flash authoring environment.

## Adding the final touches

Congratulations on creating your first Central application.

The FirstApp application has only the most basic functionality. You can add a number of features and elements to your application, including the following:

- Pods
- An agent
- Notifications
- User preferences
- Support for the Blast feature

All of these features and other examples are described in detail throughout the rest of this document.

## Taking the next steps

Now that you have successfully built and tested your first Macromedia Central application, you are ready to find out more about Central, the SDK, and the registration and publication process.

The next chapter provides an architectural overview of Central, gives a typical use scenario, and outlines your development tasks in relation to this architecture. Subsequent chapters provide details about developing full-featured applications, including debugging techniques, and tips and tricks. The last chapters of this book provide reference information for the Central application programming interface (API) ([Chapter 10, “API Reference,” on page 133](#)), and your product's XML descriptor file tag library ([Chapter 11, “The product.xml File,” on page 395](#)).

The Macromedia DevNet ([www.macromedia.com/go/central\\_dev\\_center](http://www.macromedia.com/go/central_dev_center)) also has many valuable articles on developing Central applications.

# CHAPTER 2

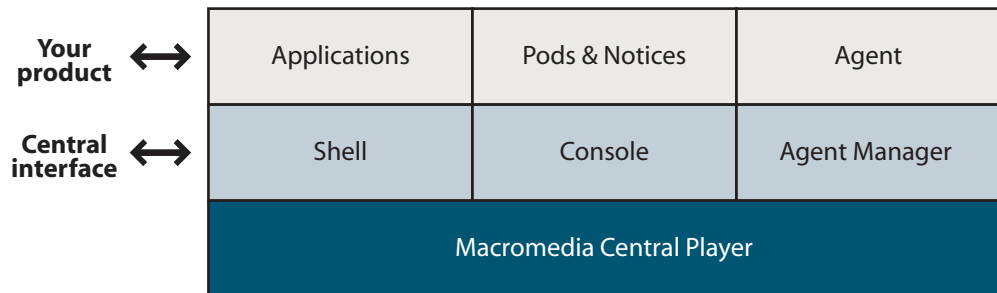
## Understanding the Macromedia Central Environment

In [Chapter 1, “Getting Started,” on page 11](#), you installed the Software Development Kit (SDK) and tested your Macromedia Central development environment with the FirstApp application. This chapter describes the Central architecture and the flow of a typical user scenario. The chapter also explains the tasks involved in creating a Central application.

### About the Macromedia Central framework

In Central, a *product* is a collection of SWF files that define the user interface and data management for your application. The FirstApp application, which is described in [Chapter 1, “Getting Started,” on page 11](#), is a simple Macromedia Flash application that runs as a product on the Central platform.

The basic architecture for the Central platform has three layers, as the following figure shows:



*Macromedia Central architecture*

The Central architecture includes the following layers:

- The Central Player displays the Central application window (often called the shell) and the Console. The Central Player sits on top of the operating system and makes network calls through the local network connection directly to remote services. It also installs and uninstalls products, and accesses the live data for the Application Finder (a built-in Central application that lists all Central applications available through Macromedia). The Central Player manages configuration information, local caching, simple data typing, and the routing of function calls.

- Applications, which appear in the shell, and pods (see “Pods” on page 23), which appear in the Console, provide users with a way to interact with products.
- Agents are SWF files that do not have user interfaces. The agents run in the background and communicate with your applications and pods through the Local Service API. The Agent Manager controls activation and deactivation of an individual agent.

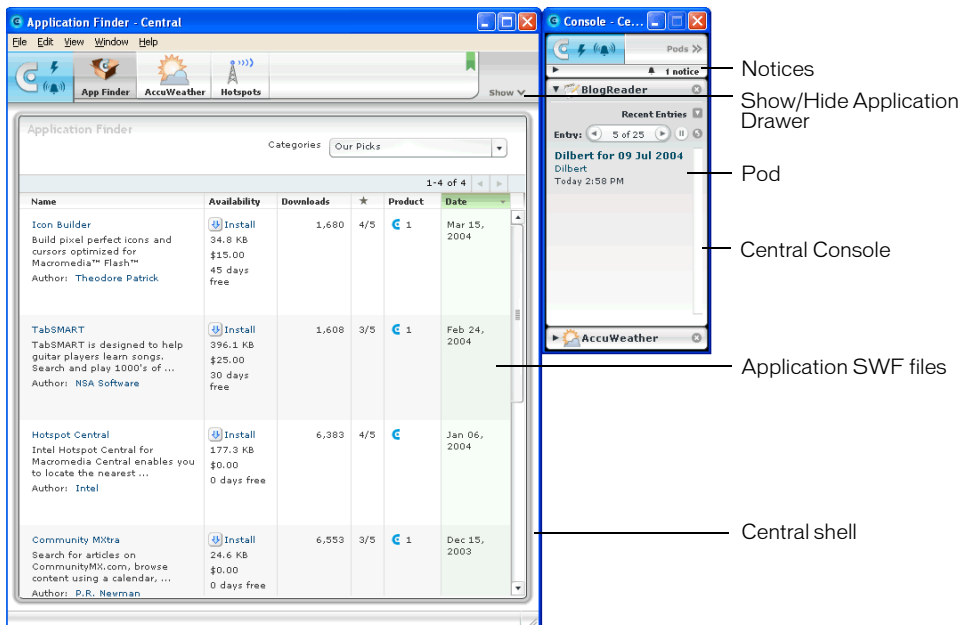
Products that you create are managed by the shell, Console, and Agent Manager. Your products may consist of an application SWF file, one or more pod SWF files, and an agent SWF file. You declare these SWF files in a product.xml file that Central uses when a user starts and deploys your product.

Your application interacts with the Central Player, shell, Console, and Agent Manager. This chapter focuses on the specific parts of your product and the way they interact with these Central services.

## Elements of the Central environment

The Macromedia Central environment consists of three main elements. These elements provide the graphical and programmatic interfaces of Central.

- The Central shell acts as a container for application SWF files and provides some user controls that are common to most Central applications. For more information, see “The Central shell” on page 21.
- The Console acts as a container for notices and pods. For more information, see “The Console” on page 22.
- The Agent Manager runs agent SWF files. Agents and the Agent Manager do not have a graphical interface for the user.



For more information, see [“The Agent Manager” on page 22](#).

The following sections describe the elements of the Central environment in more detail.

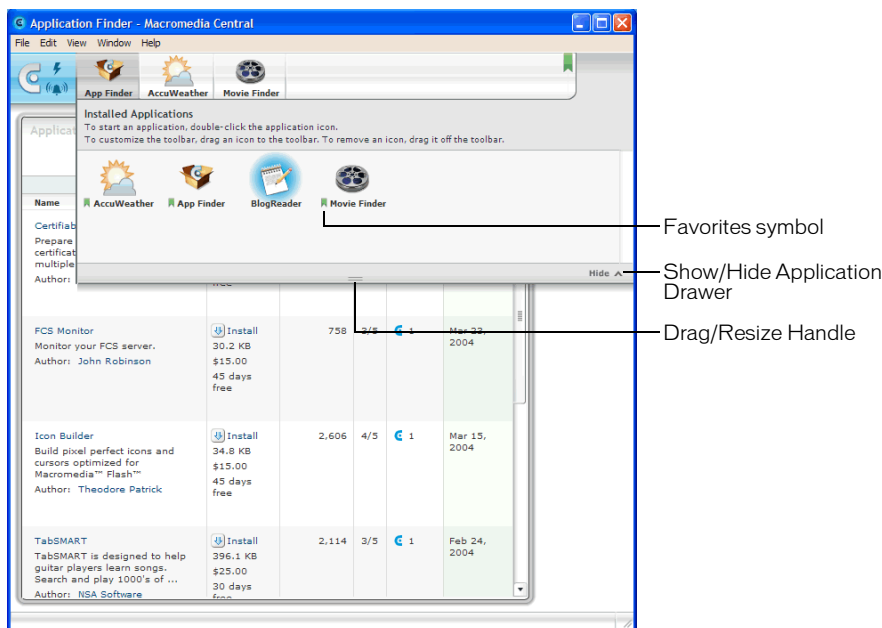
## The Central shell

The Central shell is the primary instance of the Central Player running on the user's system (the Central Player can also appear as the Console, simultaneously with the Central shell, and independently of the Central shell; see [“The Console” on page 22](#)). The top portion of the shell window contains a toolbar that shows the Central icon, icons for installed applications, and indicators for notices and network connection status. You can click the Central icon to display a list of applications that are currently installed in Central. The bottom portion of the window contains a status bar that displays a progress bar and status text, which can both be controlled by the currently running application, and the Blast UI elements. The Blast feature lets users send selected data from one Central application to another.

A user can change from one application to another within a single application window, or open multiple application windows displaying different applications. Users can open new application windows by selecting File > New Window. Each new application window is created on top of a new instance of the shell.

## Application Drawer

The Central shell contains a drop-down Application Drawer that holds icons for all installed applications. The Application Drawer lets users move icons off the main toolbar so the interface does not become too cluttered, and the drawer also holds more icons than can fit on the standard toolbar:



When the user installs a new application, the application's icon appears on the toolbar. The user clicks the Show/Hide menu option to open and close the Application Drawer. With the Application Drawer open, the user can drag the icon into the drawer to take it off the toolbar. Similarly, the user can drag an icon back on to the toolbar and close the Application Drawer.

## The Console

The Console displays pods and notices. *Pods* are small SWF files that can provide an interface for accessing the basic functionality of your application. You can use pods to provide redundant access to application features or to extend the functionality of your primary application. Pods can be displayed, collapsed, or hidden completely. Pods can only be displayed in the Console. One advantage of pods is that they are displayed as long as the Console is open, and multiple pods from different products can be displayed simultaneously. For more information, see [“Pods” on page 23](#).

*Notices* are brief text messages that any application, agent, or pod can generate to inform the user of events or new information. Notices are displayed in the Notice pane near the upper edge of the Console. For more information, see [“Notices” on page 24](#).

## The Agent Manager

The Agent Manager controls and runs agents. *Agents* are SWF files containing programming logic that the developer chooses to separate from the primary application or its pods. Implementing functionality in the agent can be useful when both an application and its pods need access to that functionality. Another advantage of running code in the agent is that the agent runs as long as Central is running, regardless of whether the associated application or pod is running. For this reason the agent is useful for monitoring remote data or performing other functions that should not be interrupted.

Each Central application can have only one agent. The Agent Manager and the agents it runs have no visible interface.

## Elements of a Central application

Your Macromedia Central application can consist of one or more Macromedia Flash SWF files, which communicate with the Central environment through its application programming interface (API). A Central *product* can contain more than one application, and an application can contain more than one SWF file, depending on how it is designed. The separate parts of an application can communicate with each other using LCService, LCDDataProvider, or localConnection objects. For more information on local connections, see [“Passing data among product parts” on page 47](#).

A Central application can contain the following elements:

- Application SWF files are Flash SWF files that provide the primary interface for the application and appear in the application window. For more information, see [“Application SWF files” on page 23](#).

- Pods are small SWF files that provide alternate versions of the application UI to display in the Console. A pod may contain a subset of functionality for the application, or alternate functionality than the presentation in the application window. For more information, see [“Pods” on page 23](#).
- Agents are SWF files that run in the background and have no user interface. Agents are a good place to put program logic that is used by both an application SWF file and a pod SWF file. For more information, see [“Agents” on page 23](#).
- Notices are messages generated by any Application, Pod, or Agent. Notices appear in the Console. For more information, see [“Notices” on page 24](#).
- Application icons appear in the user interface in the toolbar and the Application Drawer. For more information about icons, see [“Product files” on page 26](#).

## Application SWF files

Each application that you develop for Macromedia Central consists of one or more SWF files. Central application SWF files run as application windows within the shell. To operate correctly in the context of the shell, your application SWF files must support the Central application API, by containing functions in the ActionScript of your SWF files. Central calls these functions to notify your application of certain events and to request information. Because Central applications actually run inside the application window SWF file, they must also observe certain coding conventions to function properly in Central (such as not using the `_global` variable when creating objects). All of the APIs and coding conventions are discussed in detail in the following chapters.

## Pods

Pods are small SWF files that can provide redundant access to an application’s functionality or extend that functionality. Because pods are displayed in the Console, they can persist even when the user switches to another Central application. Pods can be useful for displaying frequently updated data that users might want to be continuously available on the desktop, such as stock quotes or news headlines.

For example, the pod for a weather application might show the current weather for San Francisco and let the user enter a ZIP code to see the weather for another location. Pods can also be designed as scaled-down versions of the host application. For instance, a pod for a directory application might contain a search field and display a phone number.

A Central application may have no pods, one pod, or multiple pods. Users can choose to show or hide individual pods, or the entire Console.

## Agents

An *agent* is a SWF file that runs on top of the Agent Manager. The Agent Manager and its agents have no visible interface. You should use agents to implement programming logic that is common to both an application and its pod, or that needs to run continuously when Central is running. Typical uses for agents include monitoring remote information on a server and coordinating data transfer between an application and its pods.

An agent can generate notices and can communicate events to the shell and the Console. Users can set the preferences in Central to enable or disable agents. An agent must implement the Central Agent API, as described in [Chapter 5, “Creating an Agent,” on page 93](#) and in [Chapter 10, “API Reference,” on page 133](#).

## Notices

A *notice* is a dynamically generated text message that provides information for the user. An application, a pod, or an agent can generate notices. You can set a notice to require a response from the user, or the notice can dismiss itself (requiring no user interaction) once certain criteria are met. You can show a summary of recent notices in the Console. For more information, see [Chapter 6, “Creating Notices,” on page 101](#).

## The Macromedia Central product user experience

Although a product can consist of several applications, the example described in this section is based on a product that has only one application. The example reflects how users interact with a typical application and pod.

In a typical user scenario, the following events occur:

**The user accesses your product** After you register and publish your application with Macromedia, it is available to the user on your Internet or intranet website. The user can install your application through the Macromedia Central Application Finder or directly from your server. Because you must register the application with Macromedia, the files that the user installs must match the description and location you originally provided to Macromedia. In this way, users are assured that the files they are about to install are the ones you intended.

**The user installs your product** To make your application available to users, you must provide a Flash button with some ActionScript for installing your application from your website. When the user clicks the installation button you provide, your application installs and runs in Central. If Central is not yet installed when the user clicks your Flash button for your application, the user’s existing Flash Player installs Central before installing the chosen application.

In the FirstApp example described in [Chapter 1, “Getting Started,” on page 11](#), the installation button is in the installer SWF file. You usually place the installer file on an HTML page and publish it on your web server.

**The application or user opens pods** If your application uses pods, Central can open pods automatically when the user starts the application, or the application can open pods in response to user input.

**The agent updates data** The agent receives updates of remote and local data, and sends these updates to the other parts of your application.

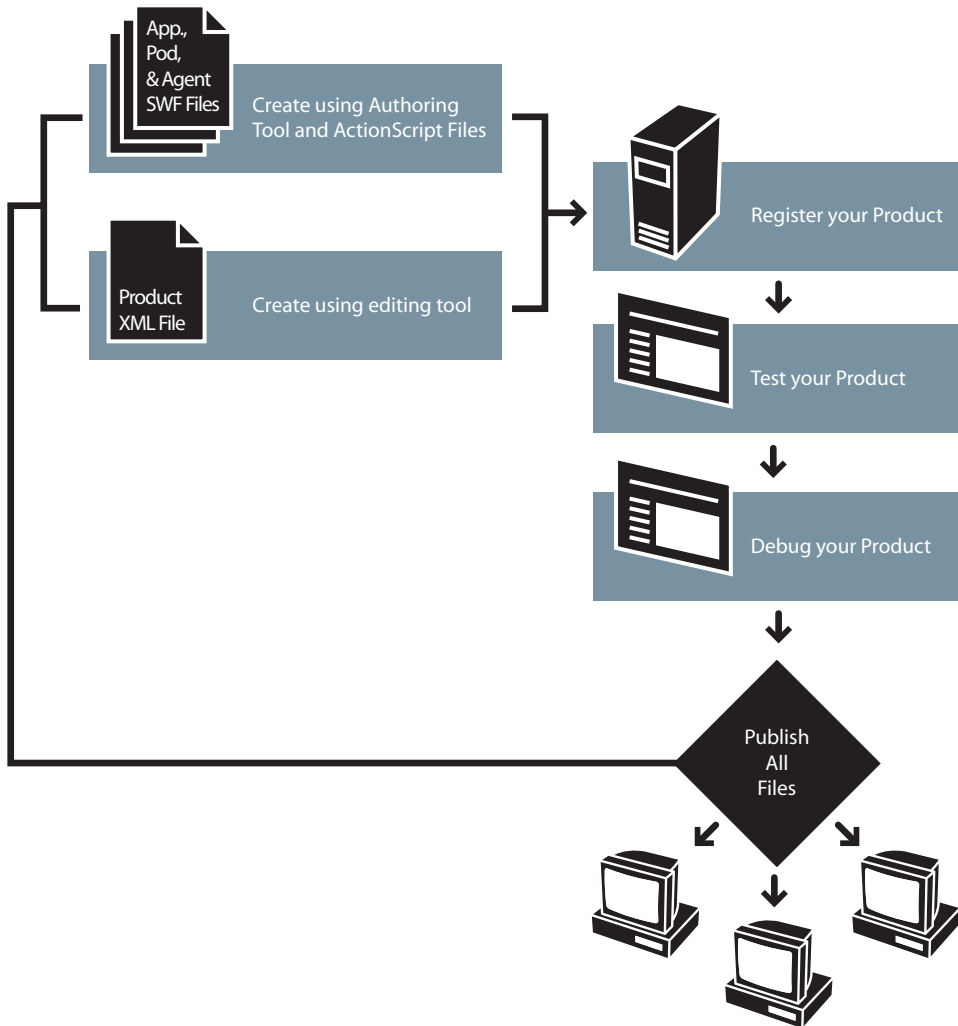
**Notices are generated** When new data is received, the agent, application, or pod can generate a notice to alert the user.



## Central development workflow

When you develop a Macromedia Central product, you typically follow these steps:

1. Create an application SWF file, and possibly additional SWF files for pods and an agent.
2. Describe the files in a product.xml file. (For more information, see [Chapter 11, “The product.xml File,”](#) on page 395.)
3. Register the product on the Macromedia website.
4. Test your product in the Central environment, debugging, redeploying, and re-installing your files until you’re satisfied with the finished product.
5. Publish your product in a location available to Central clients.



*The Central development workflow*

## Product files

A typical Central product has many parts, and these parts reside in a number of different files. To deploy your application, you will need to make these files available on a web server. Some of these files will be used to tell Central how to install your product, while the rest of the files will actually be installed on the user's computer. This section explains the use of each of these files.

**Installer files** Before you can install an application in Central, some specific code must be executed. You can find this code in the `CentralInstall.as` file included with the Central SDK. A typical way to implement this code is to include it in a simple Flash file containing a button that users can click. By setting the button to call the code in the `CentralInstall.as` file, you can create a simple installation button. The installer code works with the user's existing Flash Player to install and start Central if necessary, and then install and start your application in Central.

The only file read during download is `product.xml`, which must correctly describe all the files that will be installed. If the `product.xml` file does not exist, is corrupted, or is missing necessary information, the download fails. For more information about the requirements of the `product.xml` file, see [Chapter 11, “The product.xml File,” on page 395](#).

**Flash files** Your completed application, consisting of SWF, JPG, HTML, and other application-specific files, resides on the server that is accessible to the user. The application's `product.xml` file must be placed in the same domain as these files.

After the files are posted, you register and publish the application at the Macromedia Central Application Finder website so users can find it. The Application Finder website is integrated into the Central interface. When users click the App Finder application icon, they will see a list of applications available through the Application Finder website. When publishing your application, you can place an installation button on your site, list the application in the Application Finder, or both.

**The product.xml file** Central uses the `product.xml` file to locate and install the application, as well as the pod and agent SWF files for your application. The `product.xml` file also includes the name of the author of the application, its price, the length of the trial period for try-and-buy products, and other information needed to install the application and complete a purchase or try-and-buy transaction.

The `product.xml` file must use a specific XML tag structure and must reside in the same server directory as the SWF files for your application.

The XML schema of the `product.xml` file is described in [Chapter 11, “The product.xml File,” on page 395](#). To see a typical XML file, see [“Sample product.xml file” on page 395](#).

**Application icons** In addition to its primary SWF files, each Macromedia Central application should include one or more icons. These icons should also be SWF files. These SWF files cannot animate or execute ActionScript. You can include just one icon and let Central resize it automatically for display in different locations, or you can include multiple sizes of your icon and Central automatically chooses the most appropriate size for each display location. If you include multiple sizes, you must declare each icon size in the `product.xml` file.

**Note:** Center your icons in the graphics file or they will appear distorted in the Central interface.

If you include only one icon, it should be 35 x 35 pixels. An icon with these dimensions looks clear when resized for each of the locations where Central displays it.

## Pod files

Location	Size of icon (pixels)	Description
Standard toolbar and Application Drawer	35 x 35	To prevent distortion, Central sizes the larger dimension of your icon to 35 pixels, and scales the other dimension.
Small toolbar	23 x 23	The icons are not affected by the length of the application name. The application button size is affected, but the icon is not.
My Applications	25 x 25	These icons appear in the My Applications data grid.
My Applications details	50 x 50	These icons appear in the section that shows the application details.

**Flash files** If your product includes one or more pods, you must include these files in the directory where you posted your application file. You also must describe the pod files in the product.xml file.

**The product.xml file with pod tag** If an application includes a pod, the product.xml file should contain the pod tag, as in the following example:

```
<pod
  name="Stock Watcher"
  src="pod/pod.swf"
  enabled="1">
```

## Agent files

Typically, applications use an agent to manage data and communicate it to applications and pods that are members of its application group. Application parts can communicate with one another through LCService, LCDDataProvider, or localConnection objects used in the Agent files. An application can have only one agent, but you can make that single agent perform any number of tasks.

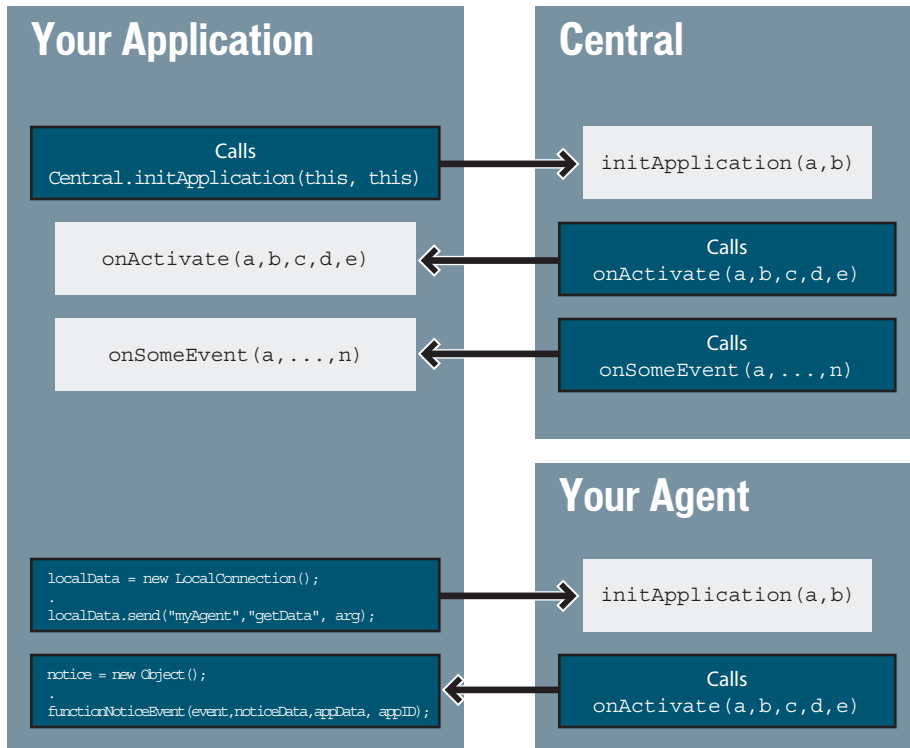
**Flash files** You create an agent as a Flash (SWF) file. Because Central plays the agent Flash file in the background, executing its code but never displaying it, the agent Flash file does not need any visual component.

**The product.xml file** You associate the agent with your application in the product.xml file by adding the agent as a subelement of the application tag, and declaring the name and the link to the agent SWF file. In the following example, the src link can be relative to the product.xml file on your server. The agent.swf file is placed on the server in the same directory as product.xml:

```
<agent name="MyAppAgent" src="agent.swf" started="true"/>
```

## Programmatic flow of a typical product

Your product consists of application, pod, and agent Macromedia Flash files, a product.xml file, and an icon file. When the user installs your product, Macromedia Central starts your Flash files, calls the `onActivate` event handler of the application SWF file, and initializes your application according to the code that you provide in the `onActivate` event handler. Central also calls event handlers for any pods and the agent. Your application, pods, and agent should provide event handlers for activating the application, getting online or offline status, and other events. Also, it is a good practice to implement the agent as the sole place where remote data is accessed and managed. This prevents situations in which the pod and application are displaying out-of-sync data, for example, and allows you to avoid replicating data-management code in multiple parts of your application.



## Typical data storage and access techniques

Macromedia Central supports existing Macromedia Flash functionality, such as the XML and `loadVariables()` APIs, as well as Flash Remoting services and a set of web services APIs. Another Flash interface is the local shared object API, which provides a mechanism for transferring data between Flash files. Central also offers a local Internet cache API as a way to store most file types locally. In addition, Central provides support for native web services using SOAP and WSDL 1.1 standards, so that no application gateway or Flash Remoting is required.

## Storing data locally

If you want to store persistent data for an application, pod, or agent, Macromedia recommends using local shared objects.

### Local shared objects

Local shared objects provide the most efficient way to store temporary data while your product is running. Use local shared objects to store data for your application, pod, or agent, such as the information that is currently displayed.

Because the shared object is locally persistent, the data that you save in it remains intact when the application stops running. The next time the application runs, it can retrieve the values it saved for the shared object when the application stopped. Alternatively, you can set the shared object's properties to `null` before the application ends, so the values are cleared.

### Local file access

The File Input/Output API lets Central applications access files on the local disk. Central applications do not have complete access to the user's local files. Instead, the `FileReference` object provides access to files in the following secure ways:

- Files may be chosen by the user using the system file dialog box. This allows access to any file on the user's computer, but only with confirmation from the user.
- Files in the local Internet cache may be opened for reading and writing without any interaction from the user. New files can be created, and data can be stored in the cache. These files are generally available from session to session, but they may be deleted if the local Internet cache becomes full or if the user clears the cache.
- Files may be downloaded from the Internet and placed anywhere on the user's disk. The user is prompted to choose a location for the downloaded file.

By limiting local access to files in these ways, the user can be assured that their files are never read or written without their permission. For more information, see [“FileReference object” on page 238](#).

### Local Internet files

When a user installs Central, Central creates a local Internet directory for caching Internet files on the user's machine. You can store any file type in this directory, except for files that are considered unsafe (.ad, .hlp, .msi, .vb, .adp, .hta, .msp, .vbe, .asp, .inf, .mst, .vbs, .bas, .ins, .pcd, .vsd, .bat, .isp, .pif, .vss, .chm, .js, .reg, .vst, .cmd, .jse, .scr, .vsw, .com, .lnk, .sct, .ws, .cpl, .mdb, .shb, .wsc, .crt, .mde, .shs, .wsf, .exe, .msc, .url, and .wsh.).

Through preferences, the user can remove these files at any time. After the URL is added to the cache, Central retrieves that data from the cache rather than from the web. You can set an expiration parameter to refresh this data.

## Sharing data

**Sharing data between application parts** When you want to share data among parts of an application group, such as an application, pods, and agent, use `LCService`, `LCDataProvider`, or `localConnection` objects. For more information about using these objects, see [“Passing data among product parts” on page 47](#).

**Sharing data across applications** Central includes a feature called Blast that allows users to share selected data in one application with other applications installed in Central. For example, a user shopping for music can send data about a CD to another application that provides reviews of CDs. The Blast button appears in the status bar of the application window when a data item is selected. Central contains a global clipboard that stores recently used data, which the user can paste into another application with the Blast button. The Auto Blast option allows users to automatically broadcast the current selection to all other applications. For more information about using the Blast feature, see [“Using the Blast feature to share data across applications” on page 76](#).

## Sharing updates with the user

Use notices to send data updates through the user interface. For more information, see [Chapter 6, “Creating Notices,” on page 101](#).

## Accessing remote data

Central applications can access data on remote servers by reading XML files and calling web services over the Internet, and by downloading files from any website.

An application can read remote XML files, and their contents are available through the XML object. This enables quick integration with existing systems by providing a simple XML version of information, which many systems already support. Applications cannot simply access any XML file on the Internet, however. If the application tries to access XML files in domains other than the application's original domain, the user will be prompted for permission.

The File Reference object lets a Central application download a file from any URL and save it to a location specified by the user. The application can then read the contents of this file. For more information about the File Reference object, see [Chapter 10, “API Reference,” on page 133](#).

Central also supports native web services, using standard SOAP and WSDL protocols. Central provides the easiest user interface layer for web services, enabling very simple development of good user interfaces. For more information about web services, see [“WebService object” on page 384](#).

Applications in Central can call web services directly without requiring any additional server-side functionality, regardless of what technology is used on the server to implement these. You can attach a user interface to a web service in just three steps using ActionScript:

```
// 1. Access the web service directly
stockService = new WebService("http://www.flash-db.com/services/ws/
    companyInfo.wsdl");
```

```
// 2. Call the web service to get company info
stockRequest = stockService.doCompanyInfo(
    "anyuser", "anypassword", "MACR");

// 3. Handle the result when it returns
function stockRequest.onResult(result)
{
    stock.companyInfo = result;
}
```

You might want to transform data on the client after receiving it. Central supports native regular expressions so that you can manipulate strings and parse HTML. For more information about using regular expressions, see [“Using regular expressions” on page 75](#).

Sometimes you might not want data to be so open and readable, and Central supports secure transmission of data. You can transmit information in encrypted form using SSL with HTTPS, the same technology that browsers support. In Central, information is stored locally in decrypted form, the same as SWF files and other information on the user’s hard disk. Authentication to network services is enabled by HTTP challenges for name/password combinations, which are handled by Central itself. In this way, applications do not have direct access to a user’s authentication information. You can also securely access web services, with encrypted communication over HTTPS and authentication provided by basic HTTP challenges. Central also provides support for implementing more advanced authentication, such as WS-Security and SAML.





# CHAPTER 3

## Building a Central Application

Building an application for Macromedia Central consists of creating one or more SWF files within the Macromedia Flash authoring environment. The SWF files function as your application, pod, agent, and icons.

To make a SWF file work correctly in the scripting context of Central, you must include specific ActionScript code that allows it to communicate with the Central environment. You must add some ActionScript functions that allow the SWF file to receive events from Central and some additional ActionScript methods used to send messages to the Central environment.

This chapter describes the steps required to create a Central application and the ActionScript code that you need to use.

### Macromedia Central application development workflow

Development of a Macromedia Central application typically includes the following tasks:

- Defining the application's functionality
- Identifying the remote data, such as XML or web services, that the application will use
- Creating the main application SWF file
- Creating pod SWF files, if necessary
- Creating an agent SWF file, if necessary
- Creating icons for the application
- Creating a product.xml file for the application
- Registering the application with Macromedia and obtaining a product ID
- Deploying the application on your website
- (Optional) Listing your application in the Central Application Finder

The sections and chapters that follow explain how to complete each of these tasks.

# Migrating from version 1.0

Macromedia Central 1.5 contains a number of features and enhancements that may affect applications that you have developed for Central 1.0. The following sections describe the changes that may affect your application as you migrate it to Central 1.5.

## Flash Player 7

Central has migrated to version 7 of Macromedia Flash Player. The two primary features of the new player are ActionScript 2.0 and support for version 2 of Macromedia Components.

Flash Player 7 also includes the following features:

**Mousewheel support** Many of the components, such as the ScrollPane, will automatically respond to the mousewheel. In addition, your application can implement listeners for the `onMouseWheel` event.

**Context menus** Applications can now customize context menus. Your application can remove most of the default context menu entries, as well as create new entries with custom callback methods. For more information, see “[Providing custom context menus](#)” on page 75.

**MovieClipLoader** The MovieClipLoader class allows applications to track the loading of images and movies.

To learn more about Flash Player 7, see [www.macromedia.com/devnet/mx/flash/articles/migrate\\_flashmx2004.html](http://www.macromedia.com/devnet/mx/flash/articles/migrate_flashmx2004.html).

## ActionScript 2.0

With the migration to ActionScript 2.0, Central provides advanced object-oriented language capabilities. ActionScript 2.0 builds on ActionScript 1.0, but also introduces some changes to the core language that may affect migration of existing applications:

**Case sensitivity** All aspects of the language are now case sensitive. When existing applications are compiled with ActionScript 2.0, there may be compile-time or runtime errors if the code does not consistently refer to a variable or function name with the same case.

**Strong typing** ActionScript now supports declaration of types for variables and functions. This allows you to ensure that a particular variable holds a value of a specific type, such as Number or Object.

**New language keywords** The keywords `class`, `interface`, `implements`, and `extends` are now part of the ActionScript language and cannot be used for variable names.

The Central SDK includes a set of class definitions that can be installed into Macromedia Flash MX 2004 to support strong typing of Central specific classes. All Central classes are installed as members of the `mx.central` package.

## Version 2 components

Central 1.5 introduces support for the second generation of components, sometimes referred to as the “v2 components.” The Central SDK provides an extension package for Flash MX 2004 that includes the components available to Central applications.

These new components offer improved performance and greater flexibility in styling and behavior. The API for the components has changed, and applications must be compiled for ActionScript 2.0 to use the new API. You must address the following issues when you migrate to the new components:

**Direct access to properties** Component properties are now directly accessible using properties rather than methods. For example, to set the label property of a button, you must write `button.label = "Submit"` rather than `button.setLabel("Submit")`.

**New event model** Event listeners should be used to handle events such as mouse clicks and keyboard entry. Use the `addEventListener()` method on most Components instead of calling `setChangeHandler()`.

**Intrinsic classes** ActionScript 2.0 requires intrinsic classes to be installed in order for the compiler to be able to enforce strict types. The Central 1.5 SDK includes a set of intrinsic class definitions that must be installed in Flash MX 2004 so that the compiler can recognize the Central components when strict typing is used.

**New component names** Many of the components have changed names and have dropped the “M” prefix in their class names. For example, `MPushButton` is now `Button`. Other components have been replaced with completely different components. For example, the `MCalendar` component has been split into `DateField` and `DateChooser`. For information about updates to components, including which components have been deprecated, see “Component changes in the Macromedia Central SDK” in *Building Central Applications with Components*.

## Central 1.5 features

Some of the key new features include:

**Local file access** Central applications can now access files on the local disk. Files can be created in the local Internet cache, or users can choose one or more files for the application to use.

**File upload and download** Applications can download files from a remote server and store them locally on the disk. They can also upload files from the disk to a remote web server using HTTP `post`.

**HTTP compression** Connections to web servers now automatically support gzip compression over HTTP. This compression is a part of the HTTP protocol and does not require files to be compressed with gzip, nor will Central decompress files that are already compressed with gzip. This reduces bandwidth requirements and occurs transparently whenever a server supports it.

**Automatic network detection** In the default configuration, Central will attempt to automatically detect if the user’s computer is connected to the Internet, and update the online or offline state. Users may have relied on your application working only in online mode. The online or offline state may change without the user’s intervention. Be sure to test your application thoroughly in both online and offline modes.

## Central 1.5 changes

Some critical changes for developers include:

**Behavior of `_root`** The `_root` property no longer refers to the root of the entire Central application window. In the default case, `_root` refers to the currently loaded SWF file. This is accomplished with an additional MovieClip property, `_lockroot`. When your movie clip is loaded, the `_lockroot` property of your application's SWF file is set to `true`. When an application refers to `_root`, Central finds the highest-level MovieClip that has `_lockroot` set to `true`. Central applications should avoid the use of `_root` and `_lockroot`. (For details, see [“Central coding conventions” on page 122.](#))

**Application resize behavior** Applications cannot define a maximum screen size. The `getMaximumSize()` function has been deprecated and is no longer called by Central. Applications must implement `onResize()` to ensure that all elements are visible to the user.

**Exact domain policy** When an application tries to access a URL outside the exact hostname from which it was installed, the user is prompted to grant permission to the application. For example, if your application was installed from `http://www.mysite.com/central/`, it cannot connect to a server at `http://webservice.mysite.com/` without the user granting permission. For information about configuring the interaction of the permission dialog box, see [“Accessing information across domains” on page 77.](#)

## Initializing an application

This Software Development Kit (SDK) comes with a `Samples` directory that contains a selection of sample applications. One of these is called `StockWatcher`. The `StockWatcher` application comes with several ActionScript files. The `stock.as` file contains ActionScript code that initializes the application in the Central environment when the user selects it from the list of installed applications. The `stock.as` file also contains ActionScript functions that are needed to respond to events that the Central shell passes to the application. These functions and events are described in the next section.

To properly initialize your application and allow it to communicate with the Central shell, you must initialize the SWF file in ActionScript. The initialization depends on the type of SWF file you are creating:

- In an application SWF file, call `mx.central.Central.initApplication(appSWF, app)`;
- In a pod SWF file, call `mx.central.Central.initPod(appSWF, pod)`;
- In an agent SWF file, call `mx.central.Central.initAgent(appSWF, agent)`;

The initialization call should be at the end of the initialization code. This call informs Central that your application, agent, or pod has been loaded.

The `StockWatcher` application calls this function after it declares its variables and calls its own `initPreferences()` function, which is located in the `preferences.as` file. The `StockWatcher` application's `initApplication()` call is simple:

```
mx.central.Central.initApplication(this, stockWatcher);
```

The two parameters of the `initApplication()` method (`this`, `stockWatcher`) are references to the Flash SWF (application) instance and the callback object, respectively. The callback object is the `ActionScript` object that contains the functions Central can call to pass events and information to the application, pod, or agent. Central looks for the callback functions in this object. These functions are described in [“Implementing the application methods” on page 37](#).

When Central receives the initialization call, it responds by calling the `onActivate()` function in the callback object you specified in the `initApplication()` call. This function is the first of several that you should include in your SWF file `ActionScript` code to make it possible for Central to communicate with the SWF file. The next section describes these functions.

The `initApplication()` call should be the last method called as part of an application’s startup code. It is recommended that the `onActivate()` function be defined before the `initApplication()` call is made.

## Implementing the application methods

After your application registers itself with the Central environment using `Central.initApplication()`, Central begins sending it events and information. For your application to respond to these events, you must implement most methods in the `Application` object in your application’s `ActionScript` code. The entire set of Central methods is described in [Chapter 10, “API Reference,” on page 133](#). If you use `ActionScript 2.0` to implement the `mx.central.Application` interface, you must declare all methods described in the following table.

A well-behaved Central application contains the following methods of the `Application` object:

Function name	Description
<code>onActivate()</code>	Called by Central each time an application is displayed in the application window. The function passes the callback object and other initialization information to the application.
<code>onDeactivate()</code>	Called by Central each time the user shuts down the application by selecting a different application, closing the application window, or quitting Central.
<code>onNetworkChange()</code>	Called by Central whenever the user changes the online or offline status of Central. Central passes this status information to the application.
<code>onResize()</code>	Called by Central whenever the Central application window is resized.
<code>getMinimumSize()</code>	Called by Central to request the application’s minimum size, in pixels. Central resizes the application window to accommodate the application if necessary.
<code>onNoticeEvent()</code>	Called by Central when a notice is dismissed.
<code>showPreferences()</code>	Called by Central when the user selects the <i>ApplicationName</i> > Preferences menu item.
<code>onUninstall()</code>	Called when the application is being uninstalled from Central.

When implementing these methods, place them at the top level of the object instance passed as the second parameter to the `initApplication()` method. In previous versions of Central, it was recommended to place these functions at the top level of your SWF file. In Central 1.5 and later, you should use `ActionScript 2.0` to implement these methods using the `mx.central.Application` interface, as described in the next section.

Many of these functions are demonstrated in the `StockWatcher` application or the `FirstApp` application provided with this document. For more information about these methods, see [“Application object” on page 171](#).

## Implementing `mx.central.Application`

`ActionScript 2.0` allows you to use strong typing to ensure that you use the API correctly. In Central, the `mx.central.Application` interface provides a set of methods that your application object must implement. By implementing this class, the compiler can check your code to ensure that you have the correct methods with the correct types before you even run your application. To implement this class, you must install the Central intrinsic classes, which are a part of the `Authoring Extensions` packaged with the SDK.

Your class must be written in a separate `ActionScript` file with the same name as the class itself. For instance, if your `Application` class is called `MyCentralApp`, you must store your class in a file called `MyCentralApp.as`.

Your class should be declared with the `implements` keyword, and should contain all the methods in the `Application` interface. For example:

```
class MyCentralApp implements mx.central.Application {

    // member variable required to store a reference to the current shell
    var gShell:mx.central.Shell;

    function MyCentralApp()
    {
        // constructor initialization here
    }

    function onActivate(shell:mx.central.Shell, appID:Number, shellID:Number,
                        baseTabIndex:Number, initialData:Object):Void
    {
        // store the shell for later
        gShell = shell;
    }

    // implement the other methods here
    // ...
}
```

After you implement the `mx.central.Application` class, you need to instantiate the class and notify Central that it is available. The following two techniques work:

**Instantiate the class with `ActionScript`** In `Frame 1` of your application, create an instance of your class using `new` and then pass this instance as the second parameter to `mx.central.Central.initApplication()`.

The following example uses the `MyCentralApp` class:

```
var myApp:MyCentralApp = new MyCentralApp();

// initialize Central, with myApp as the listener object
mx.central.Central.initApplication(this, myApp);
```

This technique allows you to pass parameters to the class constructor when the class is instantiated. It also allows your class to exist independently of any user interface.

**Attach the class to a symbol** To use this technique, you need to create a special symbol in your application SWF file that will correspond to your class.

#### To create a special symbol in Flash MX 2004:

1. Create a new symbol in your application's library with View > New Symbol. Enter a name for the symbol.
2. Right-click the symbol and click Linkage.  
The Linkage Properties dialog box opens.
3. Select the Export for ActionScript check box.
4. Enter the name of your class in both the text boxes for Identifier and AS 2.0 Class.
5. Click OK.
6. Drag the empty symbol onto the Stage.

To ensure that `mx.central.Central.initApplication()` is called when your application starts, you must add the call to the constructor for your class.

For example, the constructor for `MyCentralApp` must be changed as follows:

```
function MyCentralApp()
{
    mx.central.Central.initApplication(_root, this);
}
```

The first parameter passed to `mx.central.Central.initApplication()` refers to the application SWF file, and the second parameter is the current instance of the class.

This technique allows the Central Player to instantiate your class when your SWF file is initialized. It ensures that all your code is contained in your class file and does not require any code in Frame 1 of your application. It also prevents you from passing any parameters to your class, because Central instantiates the class automatically.

When implementing agents and pods, you can use the `mx.central.Agent` and `mx.central.Pod` interfaces.

## Using the `onActivate()` function

The `onActivate()` function is called by the shell when the application is about to be displayed in the application window. Central uses the `onActivate()` function to pass initialization information to your application, including a callback object you can use to send commands to the Central shell. Without this callback object, you cannot use any Central commands.

When Central calls your application's `onActivate()` function, it passes five parameters to it. The first is the shell callback object reference that you use as the object for all subsequent calls to the Central shell's methods. You should assign this reference to a variable so that you can use the reference for all your subsequent calls to the shell's API.

In the following example, `onActivate()` assigns the shell callback object to the variable `gShell` and then calls the shell's `setStatus()` function:

```
function onActivate(shellRef:mx.central.Shell, appID:Number, shellID:Number,
    baseTabIndex:Number, initialData:Object):Void
{
    gShell = shellRef;
    gShell.setStatus("The onActivate call has been received");
}
```

**The `appID` parameter** provides a unique ID number for the application. This ID is unique to the application but not to various instances of the application. The ID remains the same for an application across sessions in Central. By incorporating this ID into the name of a local shared object, you can use it to create local shared objects that can be shared by multiple instances of your application. You can store information, such as display state, in these local shared objects.

**The `shellID` parameter** provides a unique ID number for the shell (application window) that the application is running in. By combining the `shellID` parameter with the `appID` parameter in a variable name, you can allow multiple instances of the same application to create per-instance local connections. You can use these local connections to communicate among applications, pods, and agents. An application should not use the `shellID` parameter to create persistent local shared objects, because the value changes across sessions.

**The `baseTabIndex` parameter** is used to provide seamless tab-navigation support. You should set the tabbing order indexes on your application controls starting at this base index so that there is no conflict with surrounding application window controls.

**The `initialData` parameter** can be used to pass application-specific data to an application. To define the initial data, include an `initialData` tag with attributes within the `application` tag in the `product.xml` file. The attributes can have any name you choose, and the value of each attribute must be a string. To use a number, pass it first as a string and then convert it to a number with the `number()` method.

The following XML fragment shows how `initialData` values can be defined in the `product.xml` file:

```
<agent name="BetaAppAgent" src="agent.swf">
    <initialData foo="bar" black="white" good="evil" up="down"/>
</agent>
```

This XML code results in an object being passed to the `onActivate()` function with the following structure:

```
{
    foo: "bar",
    black: "white",
    good: "evil",
    up: "down",
}
```



Include any other initialization code necessary to display an application, including loading data, making connections, and so on, in the `onActivate()` function. Do not assume that your application will receive a `loadMovie()` event each time it is displayed.

The following `onActivate()` function is from the StockWatcher application's `stock.as` file. It starts by assigning the shell callback object to the variable `gShell`, assigning the application ID to the variable `gAppID`, and creating a unique string identifying the application by concatenating the values of `gAppName` and `shellID`.

```
function onActivate(shell:mx.central.Shell, id:Number, shellID:Number,
    baseTabIndex:Number, initialData:Object):Void
{
    gShell = shell;
    gAppID = id;
    gAppName += shellID;
    textField.tabIndex = baseTabIndex + 1
    dataGrid.tabIndex = baseTabIndex + 2
    ...
}
```

For more information about this method, see [“Application.onActivate\(\)” on page 173](#).

## Using the `onDeactivate()` function

The `onDeactivate()` function is called by the Central shell when the application is about to be shut down. This can happen when the user selects another application to be displayed in the application window, the application window is closed, or the user quits Central. You should assume that your application is going to be shut down after this method is called. The `onDeactivate()` function is a good place to save data, remove `setIntervals`, close data connections, and clear global variables. Be aware that you cannot rely on asynchronous local connection methods successfully completing if they are called in the `onDeactivate()` function.

For more information about this method, see [“Application.onDeactivate\(\)” on page 174](#).

## Using the `onNetworkChange()` function

The `onNetworkChange(connected)` function is called by the Central shell when Central detects that the network connection has been activated or deactivated. The user can also change the network connection status (from online to offline and vice versa). Include any code that you want to execute in response to the status change in this function, such as toggling variables used to determine whether to use data from the Internet or from the cache. For more information about caching data, see [“Caching data locally” on page 64](#). For more information about determining whether there is a network connection, see [“Tracking network status” on page 64](#).

For more information about this method, see [“Application.onNetworkChange\(\)” on page 175](#).

## Using the `onResize()` function

The `onResize()` function is called by the Central shell when the application window is resized. In this function, your application should call `shellRef.getBounds()` to determine the new window size and adjust its own size to fill the window exactly. The `getBounds()` method returns an object with two properties, `height` and `width`.

For more information about this method, see [“Application.onResize\(\)” on page 178](#).

## Using the `getMinimumSize()` function

Each time your application is displayed in the application window, the Central shell calls the `getMinimumSize()` function to determine the minimum size for your application. This function should return an object that contains two properties representing the minimum width and height of your application. If a user switches to your application while the application window is smaller than your specified minimum size, the Central application window expands accordingly. This function is optional. If your application does not include it, the application window is not automatically resized.

The following example uses the `getMinimumSize()` function:

```
function getMinimumSize(Void):Object
{
    var sObj = new Object();
    sObj.width = 400;
    sObj.height = 400;
    return sObj;
}
```

For more information about this method, see [“Application.getMinimumSize\(\)” on page 172](#).

## Using the `onNoticeEvent()` function

The `onNoticeEvent(event, noticeData, initialData)` function is called by the Central shell when a notice is dismissed. A notice can be dismissed by the user clicking a button in the Notice dialog box, by the timing out of the notice, or by a call to `removeNotice()`. An application, or a pod or agent associated with the application, can use the `addNotice()` command to create a notice. Use the `onNoticeEvent()` function to make your application aware of notice dismissals and to have it respond appropriately, depending on the nature and purpose of the notice.

The `onNoticeEvent()` call passes the following three parameters to the function:

- The *event* parameter describes the method of dismissal. This is an object with a *type* property that contains a string with one of the following four values:
  - `close` The user closed the notice without clicking the Engage button in the notice.
  - `engage` The user dismissed the notice by clicking the Engage button.
  - `timeout` The notice was automatically dismissed because of a timeout.
  - `remove` The application or one of its parts dismissed the notice by calling the `removeNotice()` command.
- The *noticeData* parameter contains an object that describes the properties of the notice. These properties are described in detail in [Chapter 6, “Creating Notices,” on page 101](#).
- The *initialData* parameter allows you to pass data that your application uses to respond appropriately to the notice event. The nature of this data is determined by you, the developer. If you decide to use this parameter, pass it as the last object parameter in the `addNotice()` call used to create the notice. For more information, see [Chapter 6, “Creating Notices,” on page 101](#).

The following ActionScript fragment is taken from the StockWatcher application included with this SDK. Its `onNoticeEvent()` function traces the parameters passed to the function and then checks whether the event type is "engage".

```
function onNoticeEvent(event:Object, noticeData:Object,
    initialData:Object):Void
{
    trace("onNoticeEvent: noticeData=" + noticeData + " initialData=" +
        initialData + " event: " + event.type);

    // if user clicks the Engage link in the notice, select a symbol
    if (event.type == "engage")
    {
        // add code here for selecting a symbol...
    }
}
```

For more information about notices and their parameters, see [Chapter 6, “Creating Notices,” on page 101](#). For more information about this method, see [“Application.onNoticeEvent\(\)” on page 176](#).

## Using the showPreferences() function

The `showPreferences()` function is called by the Central shell when the user selects the *ApplicationName* > Preferences menu item. In this function, you should use ActionScript calls that display a Preferences screen for your application. The *ApplicationName* > Preferences menu item is present in the Central UI only when the `hasPreferences` attribute is set to `true` in the `application` tag in the `product.xml` file. Local shared objects are a good way to store application-specific preferences data that must persist between application sessions.

For more information about the Central global preferences, see [“Working with preferences” on page 57](#). For more information about this method, see [“Application.showPreferences\(\)” on page 182](#).

## Using the onUninstall() function

The `onUninstall()` function is called when the user uninstalls the application from Central. (The user selects View > My Applications, or clicks the Central “C” logo in the toolbar, to start My Applications. Then the user clicks the Uninstall button.)

Include code in the `onUninstall()` function to clean up any local shared objects created by your application by setting the objects to zero. Central does not do this object cleanup for you. However, Central deletes files cached with either the `addToLocalInternetCache()` method or the `file` tag in the `product.xml` file, when the application that cached them is uninstalled.

For more information about this method, see [“Application.onUninstall\(\)” on page 181](#).

## Using the shell API in an application

In addition to the `initApplication()` command discussed earlier (see [“Initializing an application” on page 36](#)), there are several other commands that an application can use to communicate with the Central shell. These commands allow your application to perform tasks such as resizing, displaying information in the status bar of the application window, working with global preferences, and more.

Function name	Description
<code>requestSizeChange()</code>	Changes the size of the application window. Central responds with a call to the application’s <code>onResize()</code> method.
<code>setProgress()</code>	Displays a progress bar at the bottom of the application window.
<code>setStatus()</code>	Displays a string at the bottom of the application window.

These methods are called on the shell object reference that is passed to the `onActivate()` method in the application, as in the following example:

```
shellRef.requestSizeChange(500, 400)
```

There are also commands for working with pods, agents, and notices, as well as advanced features of Central. For more information about working with pods, see [Chapter 4, “Creating Pods,” on page 81](#). For more information about working with an agent, see [Chapter 5, “Creating an Agent,” on page 93](#). For more information about working with notices, see [Chapter 6, “Creating Notices,” on page 101](#). For more information about advanced features of Central, see the later sections of this chapter.

### Resizing the application window

Your application may need to change the size of its display to accommodate additional information or user interface elements, such as a control panel or detail view. You can resize the application window when these elements do not fit in the current application window.

To determine the current size of the window, use the `shellRef.getBounds()` command. Do not use the Stage object. Because your application appears in the application window, it does not have access to the full dimensions of the Stage. The `getBounds()` command returns an array containing integers for the height and width of the display area, respectively.

To change the size of the application window to accommodate your application’s new size, use the `shellRef.requestSizeChange(width, height)` command. Pass integers for the height and width of the area that you want the application to occupy. Be aware that the window might not be able to accommodate the requested size, depending on the screen size and other factors. The Central shell calls your application’s `onResize()` function when the resize operation occurs. You should include a `getBounds()` command in your `onResize()` function to determine the new size of the area that your application should occupy in the application window. The `getBounds()` command returns the width and height of the application area. The `onResize()` function is also a good place to include code that updates the application layout to match the new window size.

## Writing a layout manager

Central 1.5 introduces new window resizing behavior. Central no longer recognizes a maximum size for your application window. The user is able to resize the window to any size larger than the application's minimum size. If the user stretches the Central shell in one application, the shell maintains that size as the user switches to other applications (assuming that the shell size is at least as big as the application's minimum dimensions). In other words, a user may resize the Central shell larger than your application's minimum size, and then switch to your application. Your application should reposition, or resize, the user interface elements to match the new window size. To manage the resizing features of Central, your application can contain a layout manager function.

**Note:** One possible solution for handling resize changes in your application is to write the application by using Macromedia Flex, which has a built-in layout manager.

First, consider the behavior for the application when the user either resizes the application window or switches from another application with a window size bigger than your application's minimum size. Examine each screen in the application and determine how you want each area to behave as the window size changes. You may want some areas to resize and reposition themselves as the window changes, while others remain the same.

Then, once you have decided on the general behavior for your application layout, create an `onResize()` event handler that can cascade the new window size to the elements in your application.

**Note:** You may also want to call the `onResize()` function in your application's `onActivate()` function to make sure that the application layout properly fills the shell window on startup.

Finally, within each area of your application, you need to call the `setSize()` method for your components to have them resize to fit the new layout. If your application contains custom components that you need to resize, you need to write `setSize()` methods for them as well.

If changes in your application require a larger shell size, you can use `Shell.requestSizeChange()`. For information, see [“Shell.requestSizeChange\(\)” on page 370](#).

### onResize() function example

The following example from the StockWatcher application, included in the Central SDK, uses the `setSize()` method in conjunction with the `bounds` property in an `onResize()` function to adjust the component sizes:

```
function onResize (Void):Void
{
    // make sure to get bounds, rather than stage width/height
    var bounds:Object = oShell.getBounds();

    // layout the panes: the grid stretches but company info does not.
    // this means that the company info moves up and down when the
    // grid flexes.
    this.fDetailsTab.setSize(bounds.width - (this.fDetailsTab._x*2), null);
    this.fDetailsTab._y = bounds.height - this.fDetailsTab._height - 5;

    this.fFindTab.setSize(bounds.width - (this.fFindTab._x*2),
```

```

        this.fDetailsTab._y - this.fFindTab._y - 10);

// layout content in top area - move buttons around so that
// they spread out evenly when the window gets wider/narrower
var contentbounds:Object = this.fFindTab.getContentBounds();

this.fTossButton._x = contentbounds.xMax - this.fTossButton._width - 5;
this.fAlertsButton._x = this.fTossButton._x - this.fAlertsButton._width - 3;
this.fRefreshButton._x = this.fAlertsButton._x - kDefaultButtonWidth - 5;
this.fUpdated._x = this.fAddButton._x + kDefaultButtonWidth;
this.fUpdated._width = this.fRefreshButton._x - this.fUpdated._x - 5;

fResultsGrid.setSize(contentbounds.xMax - 32, contentbounds.yMax -
fResultsGrid._y - 5);

// layout details in bottom area
// just line up fDetails with fDetailsTab
contentbounds = this.fDetailsTab.getContentBounds();
this.fDetails._y = this.fDetailsTab._y + 13;
this.fDetails.fRemoveButton._x = contentbounds.xMax -
    this.fDetails.fRemoveButton._width - 5;

// reposition preferences dialog
if (fPrefs != undefined) {
    fPrefs.resizePreferences(bounds.width, bounds.height);
}
}

```

## Displaying status information

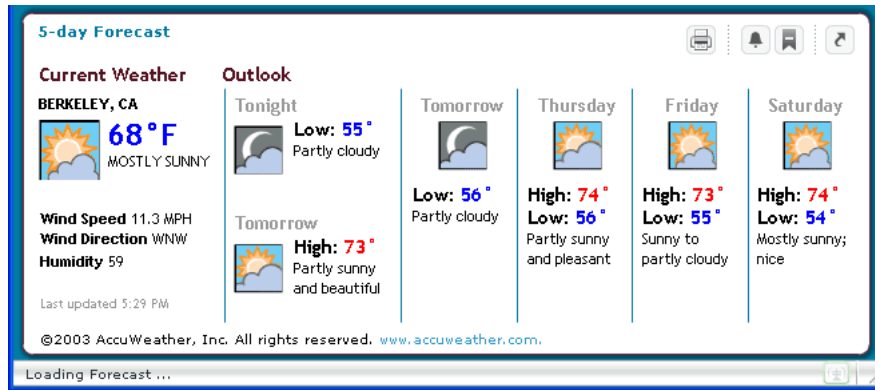
You can display a progress bar and a status message in the status area at the bottom of the application window.

Use the `shellRef.setProgress(percent)` command to display a progress bar. Set the `percent` parameter to the percentage of the progress bar that you want filled. Set the `percent` parameter to 100 or more to remove the progress bar. Set the `percent` parameter to -1 to display a barber pole animation.

Use the `shellRef.setStatus(message)` command to display a message in the status bar. Set the `message` parameter to the string that you want to display.

The following ActionScript code displays the message “Loading Forecast...” in the status bar:

```
gShell.setStatus("Loading Forecast...")
```



*The status bar displaying a message.*

To clear a status message, pass an empty string to `setStatus()`:

```
gShell.setStatus("")
```

## Passing data among product parts

If you design a Central application with multiple parts, you will probably want those parts to communicate with each other. For example, you might want the agent to gather data from the Internet and pass it to the application, a pod, or both.

There are three ways to pass information between application parts. One way is to use the `LCService` object implemented in Central to create a “client-server” relationship between one of the application parts and the others. This structure allows you to pass data by calling handlers on the “server” from the “clients” and vice versa. These calls can be made synchronously or asynchronously.

The second way to pass information between application parts is to use an `LCDataProvider` object provided by Central. The `LCDataProvider` object is for storing data that any of your application parts can access and edit. Methods used to get or edit data in an `LCDataProvider` object return results synchronously. The `LCDataProvider` object is essentially a modified version of the `DataProvider` object included with Macromedia Flash.

The third way is to create your own local connections for passing data between application parts.

## Communicating using the `LCService` object

Central provides an `LCService` object that you can use to create a virtual client-server mechanism that your application, pods, and agent can use to pass data among themselves. The `LCService` object is named for the local connection mechanism that underlies it.

The LService object allows your application, pods, and agent to define functions that associated application parts can call; for instance, you can set up an LService object that allows a pod to call a function in an agent. A typical pattern is to implement the “server” functionality in an agent and corresponding “client” functionality in the application and pod.

When using the LService object, you must define a programming interface composed of the functions that the server and client will implement. After defining the interface, you can create the object.

LService objects can be created in synchronous or asynchronous mode. In synchronous mode, function calls return results immediately, and no other code executes until the function call is complete. In asynchronous mode, function calls can return results some time after the call is made, and other code can continue to execute in the meantime. To use asynchronous mode, you must have callback handlers in place to receive the eventual results of the asynchronous calls. Depending on the types of functions you define in your interface object, you may prefer one mode over the other. Functions that take longer to execute, such those that retrieve remote data, may be more suited to asynchronous mode, whereas short functions that execute rapidly may be better suited to synchronous mode. In some cases, you may want to use more than one object in more than one mode.

The following sections describe the steps required for using the LService object.

## Defining the programming interface

The first step in using the LService object is to define the functions that you want to call on the “server,” and those that the server can call in the clients. Remember that the client-server relationship set up with an LService object is virtual. The server is simply the application part, such as an agent, that you designate to execute some centralized, server-style code. The “clients” are simply those application parts, such as the application or a pod, that you designate to call the methods of the server and respond to the results that those methods return.

To declare which methods are implemented by each side of the local connection, the LService object uses the interface object that you created. The server and all clients need to use exactly the same interface definition. Macromedia recommends that you define the interface in a separate .as file and use `#include` to include the file in every other SWF file in your application.

To do this, create a new object with the properties `name` and `interfaces`. The `name` property should be globally unique and contains the name used to identify the interface. This name is used by the client and server to determine how to connect to the other side.

The `interfaces` property contains an array containing two arrays of function names, one for the client and one for the server. These functions are defined by you.

The following code creates a new object named `ITestInterface` and defines the method `echo()` for the server:

```
// For simplicity, the following code is stored in a separate file
// named Interface.as

ITestInterface = new Object();
ITestInterface.name = "ITestInterface";
ITestInterface.interfaces = new Array();
```



```
ITestInterface.interfaces["Client"] = [];
ITestInterface.interfaces["Server"] = ["echo"];
```

Defining specific interfaces for each LCService object provides an additional layer of security, because clients can call only the server methods defined in the interface, and vice versa.

## Creating asynchronous server-side functionality

The next step is to create a “server-side” LCService object and implement the server-side functions declared in the interface object. For example, if you decide that your agent will contain the server-style functionality, you would create the server-side LCService object in the agent.

To create the server-side LCService object, use the `Central.LCService.createServer()` method.

When you create an LCService object in asynchronous mode, you need to define functions that can handle the server’s responses when they are received. This is described in [“Creating asynchronous client-side functionality” on page 50](#).

The following example, which might be found in an agent, includes the `Interface.as` file that defines the interface object and creates a server-side LCService object in the `onActivate()` handler:

```
// ServerAgent.as

#include "Interface.as"

function onActivate(agentManager:mx.central.AgentManager, agentId:Number,
    initialData:Object):Void
{
    gAgentManager = agentManager;
    gTestService = mx.central.LCService.createServer(ITestInterface, this,
        false);
}
```

The `createServer()` method takes the following three parameters:

- *interface* A reference to the interface object defined earlier.
- *callbackObj* The callback object where the server-side methods declared in the interface can be found. In this case, the callback object is the same object that contains the `onActivate()` handler.
- *bSync* An optional Boolean value that indicates whether the object is in synchronous mode. A value of `false` indicates asynchronous mode, and a value of `true` indicates synchronous mode. If this parameter is omitted, the default value of `false` (asynchronous mode) is used.

The part of your application that is acting as the server also needs to implement the server-side methods. The interface object defined earlier declared a method named `echo()` for the server. The following code implements the `echo()` function, which simply returns the string passed to it:

```
function echo(echoString:String):String
{
    return echoString;
}
```

Now you are ready to implement the client side of the LCTestService functionality.

## Creating asynchronous client-side functionality

To create the client-side functionality, you create another LCTestService object with the same interface name, this time using the `createClient()` method. Remember that you can implement client-side functionality as many times as needed, such as for an application and its pod or pods.

The following example creates a client-side LCTestService object named `gTestService` in the application's `onActivate()` handler:

```
// ClientApp.as

#include "Interface.as"

function onActivate(shell:mx.central.Shell, appId:Number, shellId:Number,
    baseTabIndex:Number, initialData:Object):Void
{
    gShell = shell;
    gTestService = mx.central.LCTestService.createClient(ITestInterface,
        String(appId) + "_" + String(shellId), this, false);

    gTestService.echo("foobar");
}
```

The `createClient()` method takes the following four parameters:

- *interface* A reference to the previously defined interface object. This must be the same reference used by the server-side `createServer()` method.
- *id* The name of the client. The server side uses this name to route the results of method calls to the correct client. In this case, a unique name is created from the combination of the application's `appId` parameter and the `shellId` parameter of the shell that contains it.
- *callbackObj* The callback object that contains the client side of the interface. In this case, the callback object is the same object that contains the `createClient()` call.
- *bSync* An optional Boolean value that indicates whether the object is in synchronous mode. A value of `false` indicates asynchronous mode, and a value of `true` indicates synchronous mode. The default is `false`. For more information about using synchronous mode, see [“Using LCTestService objects in synchronous mode” on page 54](#).

It is possible to encounter a situation in which the server side of the LCTestService object does not yet exist when the client side of the object is created. For example, this can happen if the server side is implemented in an agent SWF file and the client side is implemented in an application SWF file. If the agent SWF file hasn't fully loaded when the client-side LCTestService object is created, the client side has to wait for the server-side object to be created. However, the LCTestService object has this waiting functionality built into it, so you don't have to create it yourself. If the server side is not created within a few seconds of the client request, the `createClient()` call eventually fails.

To test the success or failure of the `createClient()` method, implement a `createClient_Result()` and a `createClient_Status()` handler.

Central calls the `createClient_Result()` handler when the client `LCService` object has successfully been created. Central passes the name of the object to the handler. This is a good place to put code that verifies the name of the object created and then initiates the use of the object. Attempting to call functions with the object before it is known to exist can cause unpredictable behavior in your application or pod.

Central calls the `createClient_Status()` handler when the `createClient()` method fails to communicate with the server side and does not generate a client `LCService` object. Central passes the name of the object that failed and information about the error to the handler. This is a good place to put code that verifies the name of the object and the error and that reattempts to create the object if appropriate.

The following example attempts to create a client `LCService` object called `gTestService`, and implements a `createClient_Result()` handler and a `createClient_Status()` handler:

```
gTestService = mx.central.LCService.createClient(ITestInterface, String(appId)
    + "_" + String(shellId), this);

function createClient_Result(client)
{
    if(client == gTestService)
    {
        gTestService.echo("foobar");
    }
}

function createClient_Status(client, status)
{
    // Couldn't connect to the agent; handle the error here
}
```

The preceding example also includes a call to the `echo()` method, which is defined as a server-side method in the interface object that was created earlier. Because the server and client sides of the `LCService` object have been implemented, calls to the defined interface can be made.

However, for the client to receive the asynchronous results returned from a call to a method on the server, it must implement a callback handler for each server-side method that will be called. These handlers must take the form `methodName_Result()`.

The following code implements a results handler for the `echo()` method that was called in the preceding `onActivate()` handler:

```
function echo_Result(echoString:String):Void
{
    trace("echo_Result = " + echoString);
}
```

The following code is the entire example from the preceding steps:

```
// Interface.as

ITestInterface = new Object();
ITestInterface.name = "ITestInterface";
ITestInterface.interfaces = new Array();
ITestInterface.interfaces["Client"] = [];
```

```

ITestInterface.interfaces["Server"] = ["echo"];

// ServerAgent.as

#include "Interface.as"

function onActivate(agentManager:mx.central.AgentManager, agentId:Number,
    initialData:Object):Void
{
    gAgentManager = agentManager;
    gTestService = mx.central.LCService.createServer(ITestInterface, this);
}

function echo(echoString)
{
    return echoString;
}
mx.central.Central.initAgent(this,this);

// ClientApp.as

#include "Interface.as"

function onActivate(shell:mx.central.Shell, appId:Number, shellId:Number,
    baseTabIndex:Number, initialData:Object):Void
{
    gShell = shell;
    gTestService = mx.central.LCService.createClient(ITestInterface,
        String(appId) + "_" + String(shellId), this);
}

function createClient_Result(client)
{
    if(client == gTestService)
    {
        gTestService.echo("foobar");
    }
}

function createClient_Status(client, status)
{
    // Couldn't connect to the agent; handle the error here
}

function echo_Result(echoString:String):Void
{
    trace("echo_Result = " + echoString);
}

mx.central.Central.initApplication(this, this);

```

## Looking at LCService in the StockWatcher application

The StockWatcher sample application included with the Central SDK contains an example of an LCService object being used to communicate between an agent and its application and pod. The StockWatcher application defines the interface object for its LCService object in a separate StockInterfaces.as class file. The StockInterfaces class is defined as follows:

```
/**
 * Method listings for the StockService (agent AIP) and StockClient interfaces.
 */
class StockInterfaces
{
    public static var name = "StockWatcherService";

    public static var interfaces = {
        Client: [
            "refreshPods",
            "setCompany",
            "setLastUpdate"
        ],
        Server: [
            "think",
            "getLastUpdate",
            "stockSearch",
            "getCompany",
            "addCompany",
            "removeCompany",
            "updateCompany",
            "addRule",
            "removeRule",
            "saveRule",
            "deleteStorage"
        ]
    }
}
```

The actual Server functions are implemented in the StockService class, and the Client functions are implemented in the StockWatcherApp and StockPod classes.

The StockService.as file contains the code for creating the server-side LCService object using the createServer() method. This code creates the LCService object in asynchronous mode:

```
oStockClient = LCService.createServer(StockInterfaces, this, false);
```

The first parameter, StockInterfaces, is a reference to the StockInterfaces class presented above.

The StockWatcher application's onActivate() function, found in the StockWatcherApp.as file, contains the code for creating a client LCService object using the createClient() method. This object then communicates with the server-side object in the agent SWF file:

```
oConn = LCService.createClient( StockInterfaces, sAppName, this, false);
```

The StockWatcher application also includes a StockPod class (defined in the file StockPod.as), which creates its own client-side LService object using code identical to that in the StockWatcherApp class:

```
oConn = LService.createClient( StockInterfaces, sAppName, this, false );
```

Once the StockWatcherApp, StockPod, and StockService objects have been instantiated, the LService based on the StockInterfaces class has one server-side object (in the StockService) and two client-side objects (one each in the StockWatcherApp and the StockPod).

The StockPod class also contains a result callback method for one of the functions in the LService interface. Initially the StockPod calls the stockSearch() method on the LService object as follows:

```
oConn.stockSearch(symbol);
```

Since the LService connection was opened asynchronously, the results from this call to the stockSearch() method will be returned to the stockSearch\_Result() method in the StockPod class:

```
function stockSearch_Result(result:StockItem)
{
    setTicker(result);
}
```

The stockSearch\_Result() method receives a result parameter which represents the value returned by the method that was invoked on the server side of the LService, which in this case was StockService.stockSearch() method.

## Using LService objects in synchronous mode

To create synchronous server-side LService functionality, you would use the same createServer() method, but specify a value of true for the final parameter, indicating that the object should be in synchronous mode. For example, if the StockService class used a synchronous LService connection, it would create the LService object like this:

```
oStockClient = LService.createServer(StockInterfaces, this, true);
```

When an LService object is in synchronous mode, you do not need to use callback handlers such as stockSearch\_Result(). In synchronous mode the result value of a method will be returned directly from the method call on the LService object.

For example, if the StockPod class used a synchronous LService connection, it could get the result of the server-side stockSearch() method directly, like this:

```
var oFoundStock:StockItem = oConn.stockSearch(symbol);
setTicker(oFoundStock);
```

## Communicating using the LCDDataProvider object

Central includes an LCDDataProvider object and a DataProviderClass object. Both of these objects implement an extended version of the DataProvider component that is included with Macromedia Flash and some Macromedia Developer Resource Kits. The newer DataProviderClass object supports all the functions that the previous DataProvider component included, and adds new functions that handle large data sets faster. The DataProviderClass component is built into Central, so including older versions of the DataProvider component in your SWF files will add unnecessary size to your application. However, if your application uses other components that use the older DataProvider component, there are no technical reasons requiring the code to be updated.

This section describes the basic use of the LCDDataProvider object. To use the DataProviderClass object, simply call `new mx.central.data.DataProviderClass()`. For information about all of the methods that the LCDDataProvider and DataProviderClass objects provide, see [Chapter 10, “API Reference,” on page 133](#).

The LCDDataProvider object stores data and uses synchronous local connections to propagate data changes to all clients that subscribe to the object. Those clients can also edit the data. For example, an agent could contain an LCDDataProvider object, and its associated application and pod could query the data in the object and make changes to it. Changes made by the application would be propagated to the pod and vice versa.

Creating the client and server parts of an LCDDataProvider object is similar to using an LCService object, but does not require you to specify methods. The LCDDataProvider object supports all the methods of the DataProviderClass object, with one addition: the `setData()` method. The `setData()` method is used to populate the object with an initial data set when the LCDDataProvider object is created. The `setData()` function takes an array or another DataProviderClass object as its parameter.

Queries and other commands sent to the LCDDataProvider object are synchronous, so they return their results immediately. No callback handlers are required.

The following sections describe the steps required for using the LCDDataProvider object.

### Creating the LCDDataProvider object

To use an LCDDataProvider object, you need to create the “server” part of the object and define a data set that the object will contain. After the object is created and populated with data, clients can connect to the object and access or edit the data.

To create the server side of an LCDDataProvider object, use the `LCDDataProvider.createServer()` method.

```
this.dpFoodList =  
    mx.central.data.LCDDataProvider.createServer("com.mydomain.food_list");
```

The `createServer()` method requires a name parameter. This same name string is used by the clients to connect to the LCDDataProvider object. Macromedia recommends that you use fully qualified names that have a high probability of being unique. This way, you can avoid inadvertently using a name that is also used by another Central application.

Next, you can add data to the object with the `setData()` method. `LCDataProvider` objects can be populated with an array or with another `DataProvider` object.

The following code creates an array and then sets the data set of the `LCDataProvider` object to the array:

```
// create an array
var foods:Array =
[ "corn",
  "grapefruit",
  "chicken",
  "soup",
  "bread",
  "coffee",
  "figs",
  "lemons"
];

// populate the LCDataProvider object
this.dpFoodList.setData(foods);
```

The `LCDataProvider` object is now ready to share data with clients.

## Creating the client side of the object

To subscribe to an `LCDataProvider` object from a separate application part, such as an application SWF file subscribing to an object in an agent SWF file, you use the `LCDataProvider.createClient()` method.

The `createClient()` method takes two parameters. The first is a unique name for the client. The second is the name of the `LCDataProvider` object to subscribe to. This name must be the same as the name used by the server side.

The following code creates a client `LCDataProvider` object in the SWF file's `onActivate()` handler:

```
function onActivate(shell:mx.central.Shell, appId:Number, shellId:Number,
    baseTabIndex:Number, initialData:Object):Void
{
    // create a unique ID from the appId and shellID
    var sClientID:String = String(appId) + "_" + String(shellId);
    // create the client side of the LCDataProvider object
    this.dpFoodList = Central.LCDataProvider.createClient(sClientID,
        "com.mydomain.food_list");
}
```

After the client side of the object is created, the client can access the data in the object and edit it.

If you are using data viewer components that use data providers, such as the `ComboBox`, `ListBox`, or other components, you can assign the component to use your `LCDataProvider` object by using the `dataProvider` property of the component.



In the following code, the `LCDataProvider` object `dpFoodList` is used as the data provider for a `DataGrid` component and a `ListBox` component:

```
this.fDataGrid.dataProvider = this.dpFoodList;
this.fListBox.dataProvider = this.dpFoodList;
```

You can also edit the data in the object directly by using the methods of the `LCDataProvider` object. For more information about these methods, see “[LCDataProvider object](#)” on page 271.

For example, you could add an item to your client-side `LCDataProvider` object when the user clicks an Add button by calling the `LCDataProvider.addItem()` method, like this:

```
this.fAddButton.onRelease = function()
{
    // add a string from the Food Name text field
    this.dpFoodList.addItem(this.fFoodName.text);
}
```

When a new item is added to the client side `LCDataProvider` object, it communicates the new data to the corresponding server side `LCDataProvider` object automatically. The `LCDataProvider` server object will then broadcast `modelChanged` events or other events to its listeners accordingly.

## Communicating among local connection objects

You can also choose to pass data back and forth among applications, pods, and agents by using regular `LocalConnection` objects. For detailed information about working with `LocalConnection` objects, see the Macromedia Flash documentation at [www.macromedia.com/go/fl\\_documentation](http://www.macromedia.com/go/fl_documentation). Additional information is available from the Flash Communication Server MX Support Center at [www.macromedia.com/go/flashcom\\_support](http://www.macromedia.com/go/flashcom_support).

Give your local connection objects unique names by constructing the name from the ID of the application and the associated shell or the ID of the pod and its associated viewer. For agents, you can use just the agent ID, since there is only one agent per application. These ID strings are passed to your application, pod, or agent when the `onActivate()` function is called.

Product parts such as pods, agents, or applications can query Central for the ID numbers of the other parts associated with their product, and use those numbers to construct the same local connection name that is being used by that part. Obtain the IDs by calling `getAgent()`, `getViewedPods()`, or `getViewedApplications()`.

## Working with preferences

Within a Central application, you can work with two kinds of preference information. There are Central preferences that apply to all the applications installed in the Central environment, and there are application-specific preferences. Central provides methods that allow you to query preference information and store your own preference information.

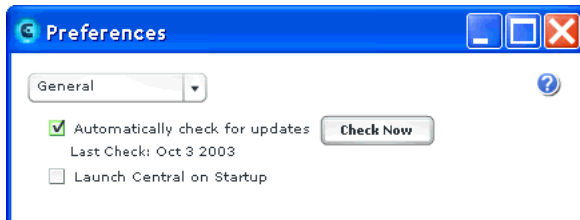
## Central preferences

The Central preferences are available when the user selects View > Preferences. The Preferences dialog box consists of three panels: General, Identity & Location, and Advanced. By default, the General panel is visible when the dialog box is first opened. The currently active panel is remembered between editing sessions. When the user returns to the Preferences dialog box after closing it or quitting Central, the Preferences dialog box displays the panel that the user viewed last.

Changes to items in the Preferences dialog box are committed when the user clicks the Done button in a Preferences panel or quits Central entirely. The user can click the Cancel button to cancel the editing operation and discard the changes.

### The General panel

The General panel allows the user to toggle the Check for Updates and Launch Central on Startup settings. When the user selects Automatically Check for Updates, Central checks for updates once each week. In addition, the user can click the Check Now button to immediately download any available updates to the Central environment.



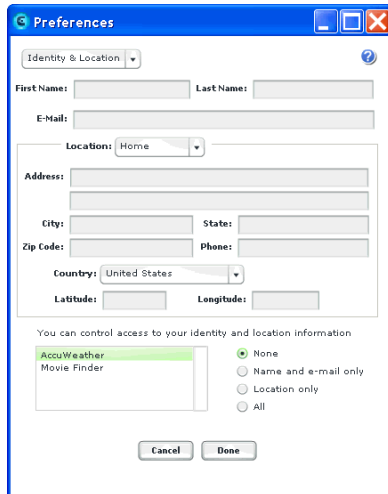
*The General Preferences panel*

### The Identity & Location panel

In the Identity & Location panel, users can view and manage their identity information (name, e-mail address, and so on) and location information (country, city, zip code, and so on). The user can also define separately named locations, and choose which applications have access to this information. The user can save any number of locations by adding and defining them individually. A location is valid even if the user does not define all the information for it.

The Identity & Location panel also includes a list of all currently installed applications. The user can select an application in the list and specify how much identity information is exposed to it. The default setting is None.

Through the Location preferences, users can allow applications access to location information. For example, a weather application might use this information to display a pop-up menu of weather forecasts for all the locations that the user has specified in the preferences.



*The Identity & Location Preferences panel*

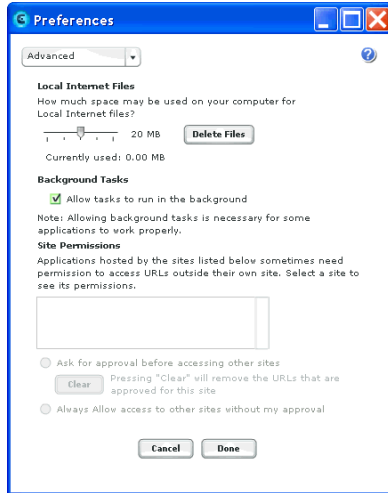
## The Advanced panel

The Advanced panel gives the user control over local Internet files, background tasks, and site permissions.

The Local Internet Files setting lets users set a global cache storage limit for applications installed in Central. This limit applies to the total space consumed by files cached by all applications installed in Central. When a user increases the cache storage limit, the new limit applies to all applications and domains. However, this limit applies only to files cached with the `addToLocalInternetCache()` method, and not to files cached at installation time with the `file` tag in the `product.xml` file.

The Background Tasks setting determines whether applications are allowed to run agents in the background. If the user turns off this setting, your agent will not be able to run.

The Site Permissions settings determine whether applications have access to URLs outside their original host domain. The list box displays the domains for applications currently installed in Central (except macromedia.com). Users can select domains from the list box and specify whether to grant them restricted or unrestricted access. Users who select the Ask for Approval Before Accessing Other Sites setting (the default) grant or deny access through dialog prompts encountered in the normal application workflow. The Access Privilege Confirmation dialog box appears when an application attempts to access the information from another domain. If the user has already granted the necessary permissions, the dialog box does not appear.



*The Advanced Preferences panel*

For more information on access privileges in Central, see [“Accessing information across domains” on page 77](#).

## Accessing Central preferences from an application

Using methods of the shell, you can access preferences and streamline the process by which users add data to their preferences. The following table summarizes the methods used to access preferences:

Function name	Description
<code>shellRef.getPreferences()</code>	Returns an object containing all the Central preferences.
<code>shellRef.newLocationDialog()</code>	Opens the New Location dialog box to allow the user to enter a new location.
<code>shellRef.editLocationDialog()</code>	Opens the Edit Location dialog box to allow the user to edit an existing location.

To access the settings that the user has chosen in Central Preferences, use the `shellRef.getPreferences()` method. This method returns a `prefObject` object that contains a property for each setting. The Preferences object has the following properties:

- `userData` An array containing `{firstName: xxx, lastName: xxx, email: xxx}`.
- `currentLocationIndex` An index that indicates the currently selected location.
- `locations` An indexed array of objects, each containing `{label: xxx, address1: xxx, address2: xxx, city: xxx, state: xxx, zipcode: xxx, phone: xxx, country: xxx, latitude: xxx, longitude: xxx}`.
- `agentsEnabled` A Boolean value that indicates whether the application can launch an agent.

The following code checks for the value of the `agentsEnabled` property and launches the agent if the value is `true`:

```
function launchAgent()
{
    prefObj = shellRef.getPreferences();
    if (prefObj.agentsEnabled = true)
    {
        shellRef.startAgent();
    }
}
```

The location data returned from `getPreferences()` is an array of locations. Each location is an object with `label`, `address1`, `address2`, `city`, `state`, `zip code`, and so on. The following ActionScript code places the location data from the preferences into a text field named `Locals`:

```
var gP = gShell.getPreferences();
var firstName = gP.userData.firstName;
var lastName = gP.userData.lastName;
var email = gP.userData.email;
var locationnum = gP.locations.length;
//outputting the location data to a text field called Locals
for ( x = 0; x < gP.locations.length; x++)
{
    Locals.text += "name: " + gP.locations[x].label + "\n";
    Locals.text += "address1: " + gP.locations[x].address1 + "\n";
    Locals.text += "address2: " + gP.locations[x].address2 + "\n";
    Locals.text += "city: " + gP.locations[x].city + "\n";
    Locals.text += "state: " + gP.locations[x].state + "\n";
    Locals.text += "zipcode: " + gP.locations[x].zipcode + "\n";
    Locals.text += "phone: " + gP.locations[x].phone + "\n";
    Locals.text += "country: " + gP.locations[x].country + "\n";
    Locals.text += "lat.: " + gP.locations[x].latitude + "\n";
    Locals.text += "long.: " + gP.locations[x].longitude + "\n";
}
```

You can help users add a new location to the location preferences by having Central open the New Location dialog box. To open the New Location dialog box, use the `shellRef.newLocationDialog(reqFields)` method. The `reqFields` parameter is an array of field names that you want to specify as required for the new location. Possible field names include `firstName`, `lastName`, `email`, `locAddress1`, `locAddress2`, `locCity`, `locState`, `locZip`, `locPhone`, `locLat`, and `locLong`.

The following code prompts the user to enter a new address, requiring only 4 of the possible 11 fields:

```
regFields = ["locAddress1", "locCity", "locState", "locZip"];
shellRef.newLocationDialog(reqFields);
```

To open the Central Preferences to the Identity & Location panel without opening the New Location dialog box, pass the string "noDialog" to the `newLocationDialog()` method. This is a good way to prompt users to grant the application access to the preferences information if they have not already done so, since the default for newly installed applications is for them not to have access to that information. For example:

```
shellRef.newLocationDialog("noDialog");
```

You can help users edit existing locations by having Central open the Edit Location dialog box. To open the Edit Location dialog box, use the `editLocationDialog()` method. The following code prompts the user to edit a location:

```
shellRef.editLocationDialog();
```

## Application-specific preferences

Central applications can store their own application-specific preferences in addition to the Central preferences discussed in the previous section. To allow the user to access your application-specific preferences, you can either provide a user interface element for that purpose in your application or enable the *Application Name* > Preferences menu item.

To enable the *Application Name* > Preferences menu item, include the `hasPreferences="true"` attribute in the `Application` tag in the `product.xml` file, as the following example shows:

```
<application name="AccuWeather" hasPreferences="true">
```

When the user selects *Application Name* > Preferences, Central calls the `showPreferences()` function in your application.

## Showing and hiding preferences

If you choose to support application-specific preferences with the *Application Name* > Preferences menu item, you should implement the `showPreferences()` function and include code in it that shows/hides your preferences interface.

One way to do this is to create a symbol with your custom preferences dialog box in it and hide/show the symbol in the `showPreferences()` function.

The following example toggles the visibility of the `fPreferences` symbol:

```
// showPreferences is called by the shell when user chooses Preferences
function showPreferences(Void):Void
{
    // toggle preferences on/off
    fPreferences._visible = !(fPreferences._visible);
}
```

The user can close the dialog box by selecting the Preferences menu item again, or you can include a button in the dialog box that allows the user to close it. If your `showPreferences()` function includes code for toggling the visibility of the dialog box, a simple way to implement a button that hides the dialog box is to call the `showPreferences()` function from within the `onRelease` function for the button.

```
fPreferences.fDialog.fDone.onRelease = function()
{
    showPreferences();
}
```

## Remembering preferences

When your custom preferences are shown, you should populate the fields with the current preference settings. When the dialog box is closed, you should store the new settings and act on any changes that the user has made. You can do this through shared objects.

In the following example, a function called `noticeMenuHandler` saves the selected item from a component to a shared object:

```
noticeMenuHandler = function(component)
{
    // remember settings
    so.data.notice = component.getSelectedItem().data;
}
```

The following code implements a `showPreferences()` function that toggles the visibility of the custom preferences dialog box, and saves or displays the current settings in a shared object named `so.data.notice`. In this case, an agent is being started or stopped.

```
// showPreferences is called by the shell when user chooses Preferences
function showPreferences(Void):Void
{
    // toggle preferences on/off
    fPreferences._visible = !(fPreferences._visible);
    if (fPreferences._visible)
    {
        // opened dialog box, set UI items to current state
        fPreferences.fDialog.fNoticeMenu.setSelectedIndex(so.data.notice);
    }
    else
    {
        // closed dialog box, act on changes
        switch (so.data.notice)
        {
            case kNoNotice:
                stopAgent();
                break;
            case kNoteInBackground:
            case kBringToAttention:
                startAgent();
                break;
        }
    }
}
```

## Tracking network status

Macromedia Central provides methods that you can use to make your application aware of whether Central is in online or offline mode. Central attempts to detect whether the network is available and notifies your application when the network status changes. The user can also change modes by clicking the lightning bolt icon in the application window toolbar. By keeping track of this information, you can adjust how your application behaves when a change in status occurs. For instance, when your application detects that the user is offline, it can retrieve information from the cache instead of from the Internet.

The Central API for tracking this information includes the functions and event handlers in the following table:

Application part	Method (you call)	Event handler (Central calls)
Application SWF files	<code>Shell.isConnected()</code>	<code>onNetworkChange()</code>
Pod SWF files	<code>Console.isConnected()</code>	<code>onNetworkChange()</code>
Agent SWF files	<code>AgentManager.isConnected()</code>	<code>onNetworkChange()</code>

Use the `isConnected()` function to determine whether Central is in online or offline mode. This function returns `true` in online mode and `false` in offline mode.

Implement an `onNetworkChange()` event handler in your application, pod, or agent SWF file to receive an event each time the user toggles the connection state of Central. Central passes a Boolean `connected` argument to the event handler that indicates the new connection state. This handler is a good place to include `ActionScript` code that responds to this change.

For example, the following `onNetworkChange()` handler could be used to check remote stock market data whenever the user goes online:

```
function onNetworkChange(connected:Boolean):Void
{
    gOnline = connected
    if(gOnline)
    {
        updateStockeData();
    }
}
```

## Caching data locally

An important feature of Central is its ability to cache data from the Internet locally. Because many users (for example, laptop users) have only intermittent access to the Internet, the ability of applications to function smoothly in the presence or absence of an Internet connection is crucial to the quality of the user experience.



By caching remote data that your application will use frequently, you can enable it to function even in the absence of an Internet connection. There are two ways to store remote data locally in a Central application. You can use local shared objects, or you can use new methods provided by Central to cache files simply by specifying the URL of the file. In general, local shared objects are good for caching previously parsed external data sources and other data structures necessary to your application. URL caching is good for saving files such as JPEG, SWF, MP3, GIF, and other external data files. Transparent GIF files are supported, but animated GIF files are not.

When you cache remote data locally with the Central methods, Flash Player in Central looks in the cache first when attempting to retrieve data with any of the ActionScript commands used for retrieving remote data, such as the `loadMovie()` command:

```
movieClip.loadMovie("http://www.myCompany.com/images/image.gif")
```

In the case of the `getURL()` method, Central does not access files in the cache, but always displays the actual live, web-based file in the browser.

Cached files are stored in the following locations:

- Windows 2000 and XP: C:\Documents & Settings\<userName>\Application Data\Macromedia\Central\#Central\<random directory>\Local Internet\<host.domainName.com>
- Windows Win 98 and ME: C:\WINDOWS\Application Data\Macromedia\Central\#Central\<random directory>\Local Internet\<host.domainName.com>
- Macintosh: HardDisk:Users:<userName>:Library:Application Support:Macromedia:Central:#Central:<random directory>:Local Internet:<host.domainName.com>

All directories under these locations are named for the domain name of the URL that was cached. Because each application is associated with a specific domain, applications cannot read data cached from domains other than their own.

From the Central Preferences interface, the user can control the amount of local disk space devoted to cache storage.

When caching files, the files are identified by their URL. Central can now distinguish between separate hosts within the same domain. For example, Central *no longer* considers the following URLs as the same:

- `http://www.mydomain.com/pub/myFile.swf`
- `http://applications.mydomain.com/pub/myFile.swf`

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

Before accessing data, using `shellRef.isConnected()` to check whether Central is in online or offline mode can help you decide whether to use cached data or to obtain fresh data from the Internet.

**Note:** Application help files cannot be cached for viewing in offline mode.

## Caching data dynamically from a URL

In Central, your applications can cache data from a URL simply by specifying that URL in an ActionScript method. There are three methods that can be used together to manage caching of remote data in applications, pods, and agents.

The following table lists the commands used by each application part to cache data:

Application part	Methods
Application	<code>shell.addToLocalInternetCache()</code> <code>shell.inLocalInternetCache()</code> <code>shell.removeFromLocalInternetCache()</code>
Pod	<code>console.addToLocalInternetCache()</code> <code>console.inLocalInternetCache()</code> <code>console.removeFromLocalInternetCache()</code>
Agent	<code>agentManager.addToLocalInternetCache()</code> <code>agentManager.inLocalInternetCache()</code> <code>agentManager.removeFromLocalInternetCache()</code>

Use the `ObjectName.addToLocalInternetCache(strUrl, bOverwrite, expiration)` method to add data from a URL to the local cache. In subsequent requests for that URL by any application, pod, or agent in Central (but not in Flash Player in a web browser), that data is retrieved from the cache rather than from the Internet. These requests can take the form of `getURL()`, `loadVariables()`, `loadMovie()`, or other similar commands. No special techniques are necessary to access cached data. It is important to realize that these types of methods always retrieve data from the cache if it is available. To get new data from the Internet, you must either delete the data from the cache with `removeFromLocalInternetCache()`, or replace it by setting the `bOverwrite` parameter of the `addToLocalInternetCache()` method to `true`.

If the `bOverwrite` argument is set to `true` and that URL's asset is already in the cache, the cached data is overwritten with new data from the URL. The cache used to store this data has a size limit set by the user in the Central Preferences. The default size is 20 MB, and that space is shared by all applications in Central. The `expiration` argument is optional and can take either a date object or an integer. If this argument is specified, the cached data is removed from the cache on the date specified by a date object, or after the number of days specified by an integer.

Most file types can be cached with `addToLocalInternetCache()`. However, some file types are considered unsafe and cannot be cached. These file types are `.ad`, `.hlp`, `.msi`, `.vb`, `.adp`, `.hta`, `.msp`, `.vbe`, `.asp`, `.inf`, `.mst`, `.vbs`, `.bas`, `.ins`, `.pcd`, `.vxd`, `.bat`, `.isp`, `.pif`, `.vss`, `.chm`, `.jsp`, `.reg`, `.vst`, `.cmd`, `.jse`, `.scr`, `.vsw`, `.com`, `.lnk`, `.sct`, `.ws`, `.cpl`, `.mdb`, `.shb`, `.wsc`, `.crt`, `.mde`, `.shs`, `.wsf`, `.exe`, `.msc`, `.url`, and `.wsh`.

It is a good idea to use `ObjectName.inLocalInternetCache(strUrl)` to check whether a prior call to `addToLocalInternetCache()` was successful and whether a URL's asset is currently in the cache. This method returns `true` or `false`. The `ObjectName` object can contain a reference to the shell, Console, or Agent Manager.

Use `ObjectName.removeFromLocalInternetCache(strUrl)` to remove a URL's asset from the local Internet cache.

## Caching data by using the product.xml file

You can specify, in your product.xml file, URLs to be cached when your application is installed. Data cached in this way is never deleted from the cache, but can be overwritten by subsequent `addToLocalInternetCache()` calls. To cache data by using the product.xml file, include a `file` tag inside the `application` tag. You can cache any file in this manner.

For example, the following code causes the file at `http://www.myCompany.com/images/shared/product_logos/logo.jpg` to be cached for the application named `Cacher App`:

```
<application name="Cacher App"
  src="cacher_app.swf"
  width="550" height="400"
  help="http://www.myCompany.com/cacher_app_help.html"
  enabled="true"
  lang="en"
  background="#FFFFFF"
  version="1"
  <icon src="cacher_app.swf" size="40"/>
  <file src="http://www.myCompany.com/images/shared/product_logos/logo.jpg"/>
</application>
```

**Note:** You can use the `initialData` tag in the product.xml file to provide initial values for variables in your application. Because these values are specified in the product.xml file, they can be easily changed without recompiling your application. This can be useful for testing purposes. However, the product still needs to be reinstalled in Central for the new values to take effect. For more information about all the tags in the product.xml file, see [Chapter 11, “The product.xml File,” on page 395](#).

Files cached with the `file` tag cannot be overwritten with the `addToLocalInternetCache()` method. The file type restrictions are the same as for the `addToLocalInternetCache()` method.

## Caching data with local shared objects

You can also cache data locally, using local shared objects. This technique is well suited to storing data retrieved from web services or other parsed data. To do this, use normal coding techniques for creating local shared objects. To ensure that your objects have unique names, include your application's unique application ID in the name of the local shared object. The unique ID is passed to your application when Central calls its `onActivate()` handler.

Local shared objects created by Central applications are stored in the application's domain directory in the Central directory. For the specific locations, see [“Caching data locally” on page 64](#).

The following code creates a new local shared object in the `onActivate()` handler:

```
function onActivate(shell:mx.central.Shell, id:Number, shellID:Number,
  baseTabIndex:Number, initialData:Object):Void
{
  theName = "sharedObj"+id
  so = SharedObject.getLocal(theName)
}
```

## Using web services

The Macromedia Central Player includes ActionScript additions that let you create applications that interact with web services on remote servers. This means that your applications can easily access remote logic and dynamic data, such as weather, stock quotes, web logs, and so on, from remote locations. This can be done in an application, pod, or agent.

Central provides a simple ActionScript interface for accessing these services that eliminates the need for you to parse the XML response data yourself. You simply create an object and call methods on that object. Your ability to use these web services depends on your knowledge of the methods they contain.

Central provides support for web services based on the Simple Object Access Protocol (SOAP) as well as XML Remote Procedure Call (XML-RPC). The ActionScript for each is very similar.

### Interacting with SOAP-based web services

This section describes the procedures used to create a SOAP-based web service object and to call the methods of the web service. Code samples are provided. For more information about each object and method, see [Chapter 10, “API Reference,” on page 133](#). The Central SOAP Web Service API supports SOAP 1.1 and WSDL 1.1.

The SOAP-based Web Service API uses a Web Service Description Language (WSDL) file, composed of XML, to create a web service object. To create a web service object, you must know the URL of the WSDL file that describes the web service. The web service object contains a parsed version of the WSDL file and allows you to call the methods of the web service directly on the object. Because each method call to a web service object is asynchronous, it returns a callback object. To handle the response to a method you call on the web service object, you define functions for the returned callback object.

#### To create and use a SOAP-based web service object:

1. Specify the URL of the WSDL file that describes the web service you are going to use:

```
var wsdlURL = "http://www.myCompany.com/services/ws/companyInfo.wsdl";
```

2. Create a new web service object with the WSDL file:

```
myWebService = new WebService(wsdlURL);
```

This causes Central to download the WSDL file, parse it, and create a web service object containing all the methods of the web service. There are two possible results of this call. If it is successful, the `onLoad()` function of the web service object is called. If it fails, the `onFault()` function of the web service object is called.

3. (Optional) Implement an `onLoad()` function for the web service object. Central calls this function when the WSDL is parsed and the object has been created. The `onLoad()` function is a good place to put code that you want to execute after the parsing of the WSDL file is complete. Central passes the WSDL document to the `onLoad()` function.

```
// optionally handle the WSDL loading event:
myWebService.onLoad = function(wsdl)
{
    wsdlField.text = wsdl;
}
```

4. (Optional) Set SOAP headers for the web service. SOAP headers are used by advanced web services to apply security, transactions, and other metadata that integrates with the SOAP operation itself. If you want to use headers, you need to provide the XML yourself. Many proposed standard SOAP headers can be found at websites such as the Web Services Interoperability Organization ([www.ws-i.org](http://www.ws-i.org)) and OASIS ([www.oasis-open.org](http://www.oasis-open.org)), in the form of specifications such as WS-Security, WS-Addressing, and so on.

You add SOAP headers with the `addHeader()` command:

```
var myHeader = new XML(headerXMLSource);
myWebService.addHeader(myHeader);
```

5. Call a method of the web service. You do this by simply calling the method on the web service object that you created. All method calls on web service objects return an asynchronous callback object, so set a variable to contain the callback:

```
// method invocations return an asynchronous callback
myMethodCallback = myWebService.methodName(parameter1, ... parameterN);
```

If you call methods on the web service before Central has finished creating the object, those method calls are queued until the object is created.

6. Define an `onResult()` function on the callback object to receive the output that is returned from the method call. Central converts the XML return value into an ActionScript object automatically.

```
// handle a successful result
myMethodCallback.onResult = function(result)
{
    // receive the SOAP output, which has been converted to an
    // ActionScript object
    myOutput = new object()
    myOutput = result;
}
```

The following is the complete example from the preceding procedure:

```
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
myWebService = new WebService(wsdlURI);

// optionally handle the WSDL loading event:
myWebService.onLoad = function(wsdl)
{
    wsdlField.text = wsdl;
}

// method invocations return an asynchronous callback
myMethodCallback = myWebService.methodName(parameter1, ... parameterN);

// handle a successful result
myMethodCallback.onResult = function(result)
{
    // receive the SOAP output, which has been converted to an
    // ActionScript object
    myOutput = new object()
    myOutput = result;
    // next, do something with the result data...
}
```

The following tables summarize the Web Service API and SOAPCall API:

### Web Service API

API element	Description
<code>new WebService(wsdlURI, logObject (optional), proxyURI (optional))</code>	<p>The <code>WebService</code> constructor requires a WSDL URL. It can optionally accept a Log object (see <a href="#">“Log API” on page 73</a>) and a proxy URL.</p> <p>The proxy URL can be used in any instance when the SOAP and WSDL traffic should be routed through some intermediary (such as a TCP monitor or management system) rather than pointed directly at the service endpoint.</p>
<code>void onLoad(wsdlDocument)</code>	Callback that is called when the web service has successfully loaded and parsed its WSDL file. Operations can be invoked in an application before this event occurs, but in this case they are queued internally and not actually transmitted until the WSDL has loaded. The parameter is the WSDL XML document.
<code>void onFault(fault)</code>	Callback that is called when an error occurred during WSDL parsing. The web services features convert parsing and network problems into SOAP faults for simple handling. The <code>fault</code> parameter is an object version of an XML SOAP fault (see <a href="#">“Error-Handling API” on page 72</a> ).
<code>SOAPCall myMethodName(param1, ...paramN)</code>	<p>Invokes a web service operation as a method directly available on the web service object. For example, if your web service has the method <code>getCompanyInfo(tickerSymbol)</code>, invoke <code>myservice.getCompanyInfo(tickerSymbol)</code>.</p> <p>All invocations are asynchronous and return a callback object. This callback object is a <code>SOAPCall</code> object that provides functions for handling results and errors on the invocation (see the following section).</p>

### SOAPCall API

API element	Description
<code>SOAPCall</code>	<p>The object type of the callback returned from all web service invocations is <code>SOAPCall</code>. These objects are normally not constructed by developers, but instead are constructed automatically as a result of WSDL parsing and stub generation.</p> <p>The <code>SOAPCall</code> object has a great number of properties and functions to handle encoding, decoding, and operation invocation that are not visible to developers. Only the public <code>SOAPCall</code> methods are described here.</p>
property XML : request	The <code>request</code> property contains the XML object that represents the current SOAP request.
property XML : response	The <code>response</code> property contains the XML object that represents the most recent SOAP response.

API element	Description
<code>void onResult(result)</code>	Callback that is called when a method has been successfully invoked and returned. The result is the decoded ActionScript object returned by the operation (if any). To get the raw XML returned instead of the decoded result, you could access the <i>SOAPCall.response</i> property.
<code>void onFault(fault)</code>	Callback that is called when a method has failed and returned an error. The <i>fault</i> parameter is an object version of an XML SOAP fault (see <a href="#">“Error-Handling API” on page 72</a> ).

## Handling SOAP web service errors

Errors can occur when you create a web service object and when you call methods on the object. When you create the web service object, you should define an `onFault()` function for the object. This function is called only if an error occurs as a result of the `new webService(WSDL)` call. This can happen if the WSDL file is not found or if it cannot be parsed successfully.

When you call a method on the web service object, you should define an `onFault()` function for the callback object. This function is called only if an error occurs as a result of the method call.

Both types of errors pass a `SOAPFault` object to the `onFault()` function that is called. The `SOAPFault` object has five properties:

- `faultcode` A string that contains a short, coded name for the error.
- `faultstring` A string that contains a human-readable description of the error.
- `detail` Application-specific information associated with the error, such as a stack trace or other details returned by the web service engine.
- `element` An XML object that represents the XML version of the error.
- `faultactor` A string that contains the source of the error. This is optional if no intermediary code is involved in the web service transaction.

The following `onFault()` function for the web service object displays the `faultstring` text in a text field:

```
myWebService = new WebService(wsdlURI);

function myWebService.onFault(fault)
{
    ErrorOutputField.text = fault.faultstring;
}
```

The following `onFault()` function is defined for the callback object returned from a web service method call:

```
myCallbackObject = myWebService.methodName();

function myCallbackObject.onFault(fault)
{
    ErrorOutputField.text = fault.faultstring;
}
```

The following is a complete example, including fault checking:

```

var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
myWebService = new WebService(wsdlURI);

function myWebService.onFault(fault)
{
    ErrorOutputField.text = fault.faultstring;
}

// optionally handle the WSDL loading event:
myWebService.onLoad = function(wsdl)
{
    wsdlField.text = wsdl;
}

// method invocations return an asynchronous callback
myMethodCallback = myWebService.methodName(parameter1, ... parameterN);

// handle a successful result
myMethodCallback.onResult = function(result)
{
    // receive the SOAP output, which has been converted to an
    // ActionScript object
    myOutput = new object();
    myOutput = result;
}

```

## Error-Handling API

API element	Description
<i>SOAPFault</i>	The type of error object returned to <code>webServiceObject.onFault()</code> and <code>callbackObject.onfault()</code> functions is a <i>SOAPFault</i> object. It is not constructed directly by developers, but is returned as the result of a failure. This object is an ActionScript mapping of the SOAP fault XML type. The following are the properties of the <i>SOAPFault</i> object.
<code>string : faultcode</code>	The short standard <i>QName</i> describing the error.
<code>string : faultstring</code>	The human-readable description of the error.
<code>string : detail</code>	The application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
<code>XML : element</code>	The XML object representing the XML version of the fault.
<code>string : faultactor</code>	In a messaging scenario where one or more applications (or intermediaries) handle or perform additional processing on the SOAP message before it reaches its ultimate destination, the <code>faultactor</code> attribute is used to identify which of these applications caused the fault.



You can log the events that take place in relation to a web service object by creating a Log object and passing the object as a parameter to the new `WebService()` constructor. When you create the Log object, you can define the types of events that the log records. You can then define an `onLog()` function on the Log object that is called each time the log receives a message. This is a good place to put code that displays the log information for debugging purposes.

The following code defines a Log object, passes it to the web service constructor, and defines an `onLog()` function that displays the log message in a field:

```
myLog = new Log("DEBUG")
var wsdlURL = "http://www.flash-db.com/services/ws/companyInfo.wsdl";
myWebService = new WebService(wsdlURL, myLog);
myLog.onLog = function(message)
{
    DebugOutputField.text += message + "\n";
}
```

**Log API**

API element	Description
<code>new Log(logLevel, logName)</code>	<p>Creates a Log object that can be passed as an optional argument to the <code>WebService</code> constructor (see <a href="#">“Web Service API” on page 70</a>). The <code>logName</code> parameter is optional, and can be used to distinguish between multiple logs that are running simultaneously to the same output.</p> <p>The <code>logLevel</code> parameter is optional. The value of <code>logLevel</code> must be one of the following. If the value is not set explicitly, the default <code>Log.BRIEF</code> is used.</p> <p><code>Log.BRIEF</code> Primary lifecycle event and error notices are received.</p> <p><code>Log.VERBOSE</code> All lifecycle event and error notices are received.</p> <p><code>Log.DEBUG</code> Metrics and fine-grained events and errors are received.</p>
<code>void onLog(message)</code>	Callback that is invoked when a log message is sent to a log.

**Interacting with XML-RPC web services**

In Central, your applications can also interact with web services based on XML-RPC (Remote Procedure Call). The technique used to interact with these services is very similar to that used for SOAP-based services. You start by creating an `RPCFactory` object. After the object is created, you can call the methods of the RPC-based service directly on the object. These method calls return callback objects that can receive the results of the method asynchronously.

**To use an XML-RPC web service:**

1. Specify the URL of the RPC web service that you plan to use:

```
myRPCUrl = "http://www.myCompany.com/services/RPC/myRPCservice" ;
```

2. Create a new `RPCFactory` object from the URL:

```
myRPCFactoryObj = new RPCFactory(myRPCUrl) ;
```

3. Invoke a method of the web service by using the `createCall()` method of the `RPCFactory` object. In the `createCall()` method, you pass the name of the method and any arguments it requires. The `createCall()` method returns a callback object that can receive the results of the web service method asynchronously. To receive those results, you define functions on the callback object.

```
myRPCCallbackObj = myRPCFactoryObj.createCall("methodName", Arg1, Arg2) ;
```

4. Define an `onResult()` function for the RPC callback object. This function is called when the results of the web service method call ("methodName" in the previous step) arrive.

```
myRPCCallbackObj.onResult = function(result)
{
    outputTextField.text = result
}
```

5. Define an `onFault()` function for the RPC callback object to receive any errors caused by the `createCall()` method:

```
myRPCCallbackObj.onFault = function(fault)
{
    debugTextField.text = fault
}
```

The following is the complete example from the previous procedure:

```
myRPCUrl = "http://www.myCompany.com/services/RPC/myRPCservice" ;
myRPCFactoryObj = new RPCFactory(myRPCUrl) ;
myRPCCallbackObj = myRPCFactoryObj.createCall("methodName", Arg1, Arg2) ;

myRPCCallbackObj.onResult = function(result)
{
    outputTextField.text = result
}

myRPCCallbackObj.onFault = function(fault)
{
    debugTextField.text = fault
}
```

The following tables summarize the XML-RPC API:

#### RPC Web Service API

API element	Description
<code>new RPCFactory(targetURL)</code>	Constructor for an RPC factory object that can generate RPC calls.
<code>RPC createCall(param1...paramN)</code>	Creates and invokes an RPC call.

## RPC Callback Object API

API element	Description
<code>onResult(result)</code>	Callback that is called when a method has successfully been invoked and returned. The result is the decoded ActionScript object returned by the operation (if any). To have the raw XML returned instead of the decoded result, you can access the <code>RPC.rpcResponse</code> property.
<code>onFault(fault)</code>	Callback that is called when a method has failed and returned an error. The <code>fault</code> parameter is an object with properties that map to the XML-RPC struct type.

## Using regular expressions

Central supports the use of regular expressions, which are groups of symbols used to match patterns within text. Regular expressions can be used to find and replace strings, remove characters from text, and test for specific conditions in text. For more information, see “[RegExp object](#)” on page 325. Also, Macromedia has useful information online about the use of regular expressions at [www.macromedia.com/devnet/central/articles/regex\\_04.html](http://www.macromedia.com/devnet/central/articles/regex_04.html).

## Providing custom context menus

Central supports the use of the Flash Player 7 `ContextMenu` and `ContextMenuItem` classes. (For details, see *ActionScript Language Reference*, or see the [online ContextMenu class Livedocs](#) and the [online ContextMenuItem class Livedocs](#).)

**Note:** Central differs from the Flash Player 7 implementation of the `ContextMenu` class in two ways: Central ignores the `ContextMenu.builtInItems` property, and Central uses its own logic to determine which `ContextMenu` objects are detected or used. The logic that Central uses for `ContextMenu` objects considers the `TextField`, `Button`, or `MovieClip` that the user clicks. If a `MovieClip` and its parent both have defined menus, Central uses the *child's* `ContextMenu` setting (Flash Player 7 does not recognize `ContextMenu` objects set in child `MovieClips`).

The `ContextMenu` class provides runtime control over the items in a Central application or pod context menu. Context menus appear when a user right-clicks (Windows) or Control-clicks (Macintosh) in the Central application or pod. By default, two variations of the Central context menu appear: the Edit menu, when the mouse is moved over a form element (the Edit menu displays items such as Cut, Copy, and Paste), or the Standard menu elsewhere (the Standard menu displays items such as Settings and Print). You can use the methods and properties of the `ContextMenu` class to add custom menu items, to control the display of the built-in context menu items (for example, Zoom In and Print), or to create copies of menus.

The standard way to modify or create a context menu consists of creating an instance of `ContextMenu`, populating it with `ContextMenuItem` objects, and assigning it to the menu property of a `MovieClip`, `Button`, or `Input TextField`. The `ContextMenu` instance can also provide a callback, which is called before the menu is shown, allowing the user to modify what is in the context menu.

## Context menu example

The following example adds a “delete” menu item to an object (myIconMC), sets some of the default menu items for that object, and provides a menu callback so that the application user can adjust the menu at runtime:

```
var itemCallback = function ( obj, item )
{
    if ( item.caption == "delete" )
        obj.deleteSelf ();
};

var menuCallback = function ( obj, menu )
{
    if ( obj.status == "locked" )
        menu.customItems[0].enabled = false;    else
        menu.customItems[0].enabled = true;    };

var menu = new ContextMenu ();
var item = new ContextMenuItem("delete", itemCallback);
menu.customItems.push ( item );
menu.onSelect = (menuCallback);

myIconMC.menu = menu;
```

**Note:** Avoid complex computations in the `onSelect()` callback function. The faster the `onSelect()` callback function returns a value, the less delay the user may notice.

## Using the Blast feature to share data across applications

The Blast feature of Central provides a way for applications to send user-selected data to other applications installed in Central. Applications can be designed to listen for this data and respond to it in useful ways.

For example, a user might look up a concert in an entertainment calendar application and then select the event data in the application. The user can then select a map application from the Blast menu to send the selected data to the map application to get directions. The user might also choose a retail application from the Blast menu to find products related to the performer, such as concert T-shirts.

Another example might be a movie finder application, in which the user selects a specific movie. The movie finder application could pass the address of the movie theater to a restaurant finder application to find restaurants near the theater. The restaurant finder could then pass the address of a selected restaurant to a weather application to find information about the weather for that location.

The Blast feature can be implemented in an application to enable the application to send data, receive data, or both. For detailed information about implementing the Blast feature in an application, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Accessing information across domains

Flash Player, as it is installed in most browsers, enforces a security sandbox that limits access to URLs from other domains. Central defines some special rules about when and how that sandbox is expanded to allow access to external domains from the Central Player. It defines a cross-domain access facility embedded in the Central controller that monitors the degree of trust a user has granted to a URL for a specific domain. Users grant applications trust through dialog boxes that pop up when applications attempt to access URLs from other domains.

### Usage scenarios

The following are some usage scenarios that call for access to URLs from other domains:

- An XML news reader that needs access to websites beyond its own domain to get feeds
- A blogger application that needs to use XML-RPC against blog sites all over the web
- A portal application that brings resources (web services, images) together in one application
- Flash Remoting sites that can be accessed across domains

### Cross-domain access rules

Resources are accessible on a per-URL basis at the discretion of the user: The user explicitly allows access through interaction with a dialog box explaining the application's request for additional permissions. The user has the following options:

- Grant access to this URL for all subsequent requests in the future.
- Grant access to this and all other URLs requested by this domain.
- Deny access to this URL.

To see and change the cross-domain access granted to a domain, the user opens the Advanced Preferences dialog box (View > Preferences). The preferences indicate whether a domain is trusted partially (trust on a per-URL basis) or completely (trust all URLs). The user can switch between the two settings. When the user changes the setting from trust a single URL to trust all, Central discards all the previously trusted individual URLs (that is, the trusted state is not remembered if the user switches back to deciding trust on a per-URL basis).

Cross-domain access preferences information persists across Central restarts by means of a shared object that Central creates. When the user uninstalls all the applications from a domain, Central discards its cross-domain access information.

### Cross-domain access settings

The Site Permissions setting in the Advanced panel of the Central Preferences dialog box allows the user to set cross-domain access. From this panel, the user can grant an application access to information from URLs outside its own domain. For example, an application from the website [www.sportsnews.com](http://www.sportsnews.com) might need access to [www.football.com.br/southamericancup/brazilcolombia](http://www.football.com.br/southamericancup/brazilcolombia) to provide the latest scores from a match.

To maintain the highest level of security, the user is advised to keep the default setting—that is, to require applications to ask for approval before accessing URLs in an outside domain. The user can then approve or deny access to outside URLs on a case-by-case basis. Central remembers the URLs that the user has approved until they are cleared in the Central Advanced Preferences. The user cannot see this list, but Central automatically checks the approved URLs when an application tries to access a URL.

If the user selects the Always Allow option, the application can access URLs outside its own domain without the user's explicit approval.

## The Cross-Domain dialog box

If the Site Permissions setting is set to the default behavior, the user sees the following message when an application wants access to information from a URL outside its domain:

“An application wants to access information from ‘[site].’”

At this point, the user has the following choices:

- The user can allow access to this URL. Central then remembers this URL and won't ask about it again in the future.
- The user can select Always Allow. In the future, this application always has access to any URLs outside its own domain.

**Note:** The user can update these choices from the Central Advanced Preferences.

- The user can select Deny. Central then denies the application access. The application must respond appropriately when the user denies cross-domain access.

For example, suppose that the user requests streaming data that your application gets from another domain, but then denies your application access to that domain. Your application should notify the user that the requested data is not available because the user did not allow access to it.

The following example loads an XML file. When the user denies access to the URL, `XML.onLoad` is invoked and the success parameter is `false`.

```
var myXML = new XML();
myXML.onLoad = function(success)
{
    if (success == false)
    {
        trace("Could not load XML");
    }
}
myXML.load("http://someSite/foo");
```

When you use the `WebServices`, `LoadVars`, `XML`, or `Flash Remoting ActionScript` methods to request a resource from another domain, users of your application are presented with the Cross-Domain Access dialog box. Any other methods that you use to request a resource from another domain are bound by the same rules as web-based Flash Player.

## Bypassing the Cross-Domain dialog box

Applications that load data from multiple domains (including subdomains) force Central to gain permission from the user to cross domain boundaries. This can be an inconvenience for applications that load data from many or varying sources (news aggregators, for example). To help developers write applications that use data from other domains, Central supports cross-domain policy files as implemented in Flash Player 7 (see “About allowing cross-domain data loading” in *Using Flash*). Policy files permit applications to load data from multiple domains, while preserving strong security rules.

**Note:** After a policy file is loaded and grants access to a domain, Central continues to use the policy file until the user shuts down Central. When the user restarts Central, Central rereads the policy files.





# CHAPTER 4

## Creating Pods

Macromedia Central applications can include small displays called *Pods*. Pods can be miniature versions of an application or partial displays of information provided by an application that appear in the Central Console. Some examples of pod functionality include the following:

- A stock market application that displays continuously updated stock quotes in a pod.
- A weather application that displays auto-updating weather maps or rainfall levels in a pod.
- A directory application that displays a pod from which the user can search for and display names and phone numbers.
- A shopping application that displays a pod with an updated list of items in a user's shopping cart.

Pods can be loaded into the Console by ActionScript in an application or agent, or automatically when Central starts.

To function properly in Central, your pod must implement the Central pod API so that it can communicate with Central and respond to event messages generated by Central. These APIs are described in the sections that follow.

### Creating a pod

Adding a pod to your Central application requires you to build a SWF file for the pod, declare it in the product.xml file, and implement some functions to initialize the pod and enable Central to communicate with it.

### Building a pod

To build a pod, you create a SWF file for the pod. The pod SWF file must be 170 pixels wide and can be from 20 to 1000 pixels high. The default height is 100 pixels.

For a good user experience, make sure your application design minimizes the possibility that its pods will push other pods off the Console. You can do this by keeping the pod height modest and by not opening excessive numbers of pods.

## Displaying a pod in Central

Pods can be added to the Console in two ways. A pod can be added automatically when its application is installed, and pods can be added programmatically while your application is running.

### Displaying a pod when its application is installed

To configure a pod to open when your application is installed, use the `pod` tag in the `product.xml` file. The `pod` tag lets you specify the name of the SWF file to be used for your pod. The presence of the `pod` tag adds the pod to the Central internal list of pods available to your application. This list of pods appears in the pod menu in the Console. The user can choose your pod from this list.

For the pod to be displayed when the application is installed, the `pod` tag must include a `viewed` attribute set to `true` (`viewed="true"`). This attribute tells Central to display the pod immediately when it is added to the list. If the `viewed` attribute is set to `false`, the pod is not displayed in the Console but is accessible to the user in the Console pod menu.

The following XML code fragment demonstrates the correct use of the `pod` tag within the `application` tag in order to display the pod when the application is installed:

```
<application name="Sample"
  src="sample.swf"
  width="550" height="400"
  enabled="true"
  lang="en"
  background="#FFFFFF"
  version="1"
  <icon src="icons/sample_24.swf" size="24"/>
  <pod name="Sample Pod"
    src="pod.swf"
    height="200"
    viewed="TRUE" />
</application>
```

If you do not want the pod to be accessible in the Console when your application is installed, you can omit the `pod` tag and use only the `podClass` tag described in the next section.

### Displaying pods programmatically

An application can be designed to display pods programmatically in response to an event, such as a user interaction. There are two ways of displaying pods programmatically that correspond to the two ways of specifying pods in the `product.xml` file.

A pod that has been specified with the `pod` tag in the `product.xml` file can be displayed by calling the `viewPod()` method. A pod that has not been specified with the `pod` tag, but has instead been specified with the `podClass` tag, can only be displayed by calling the `addPod()` method and then the `viewPod()` method.

The main difference between pods that are specified with the `pod` tag and the `podClass` tag is that pods specified with the `podClass` tag are not available in the Console pod menu or to the `viewPod()` method until `addPod()` has been called. However, using the `addPod()` method provides an opportunity to customize the pod by passing parameters to the method. You can customize the `height`, `title`, and `initialData` properties of the pod.

For example, you might design a pod that displays the scores of all the hockey games being played today. You might also have a `podClass` named `gameDetail` so the user can open up pods for each of the games she wants to track. Each one could be named for the teams that are playing, and display details for that game, through the use of the `initialData` property of pod objects.

The `addPod()` method creates a pod from a SWF file that is specified in the `podClass` tag. Once created, the pod can be viewed with the `viewPod()` method.

#### **To display a pod that has been specified with the pod tag:**

1. Call the `getPods()` method to get the pod ID for your application's pod.

The `getPods()` method returns an array of objects representing the pods that have been added by the application or its associated agent or pods. Each object contains the following properties:

- `id` A number indicating the internal ID assigned by Central to the pod. This number is read-only and is assigned by Central rather than the developer.
- `name` A string indicating the name of the pod as defined by the `name` attribute of the pod tag or the name passed to the `addPod()` method.
- `className` A string indicating the name of the SWF file for the pod as defined by the `src` attribute of the pod tag.
- `height` A number indicating the height of the pod in pixels as defined by the `height` attribute of the pod tag.
- `initialData` An optional object containing properties defined by the developer. This object can be defined and then passed to the `addPod()` method. For more information about the `addPod()` method, see the next section.

The following ActionScript returns the list of pods associated with the application, and sets the variable `currentID` to the ID of the first pod in the list. The `getPods()` method is being called by the application on the Shell object.

```
podList = shellRef.getPods();  
currentID = podList[0].id;
```

2. Call the `viewPod()` method to display the pod.

After the ID of the pod has been determined, the `viewPod()` method can be called to display the pod in the Console. The `viewPod()` method displays a pod that has already been added to the Central internal list of available pods. The `viewPod()` method requires the pod ID parameter that was returned by the call to the `getPods()` method.

The following ActionScript calls the `viewPod()` method on the Shell object and passes it the pod ID returned from the previous call to the `getPods()` method.

```
shellRef.viewPod(currentID);
```

The `getPods()` and `viewPod()` methods can be called on the Shell, Console, and AgentManager objects.

**To display a pod that has only been specified with the podClass tag:**

1. Define a `podClass` tag for the pod. The `podClass` tag includes a `name` attribute that allows you to assign a name to the pod and a `src` attribute that allows you to specify the SWF file to be used for the pod.

The following XML fragment demonstrates the use of the `podClass` tag within the `application` tag:

```
<application name="Sample"
  src="sample.swf"
  width="550" height="400"
  enabled="true"
  lang="en"
  background="#FFFFFF"
  version="1"
  <icon src="icons/sample_24.swf" size="24"/>
  <podClass name="samplePod" src="samplePod.swf" />
</application>
```

2. Call `addPod()` to add the pod to the Central list of available pods for the application.

The `addPod()` method takes an object as a parameter. This object should have the same structure as the objects in the array returned by `getPods()`. These objects have the following properties:

- `name` A string indicating the name of the pod. This is the name that will appear in the pod's title bar.
- `className` A string indicating the name of the class for the pod. This should be the same as the string defined by the `name` attribute of the `podClass` tag.
- `height` A number indicating the height of the pod in pixels. You can set a default height for the pod by including a `height` attribute in the `podClass` tag.
- `initialData` An optional object containing properties defined by the developer. This object can be defined and then passed to the `addPod()` method. For more information about the `addPod()` method, see the next section.

When the `addPod()` method is called, Central adds an `id` property to the object that describes the pod. After `addPod()` has been called, the `id` property can be accessed as a property of the object returned by `getPods()`.

The following ActionScript defines an object containing the properties of a pod, and then passes the object to the `addPod()` method:

```
var newPod:Object = new Object();
newPod.name = "samplePod";
newPod.className = "samplePod";
newPod.height = 200;

var currentID:Number = shellRef.addPod(newPod);
```

3. Call the `viewPod()` method to display the new pod.

The `viewPod()` method can be called to display the pod in the Console. The `viewPod()` method displays a pod that has already been added to the Central internal list of available pods. The `viewPod()` method requires the pod ID parameter that was returned by the call to the `addPod()` method.

The following ActionScript calls the `viewPod()` method on the Shell object and passes it the pod ID returned from the previous call to the `addPod()` method:

```
shellRef.viewPod(currentID);
```

The two different approaches to displaying pods can be used together as well. By using both the `pod` and `podClass` tags, you can implement creative ways of using and displaying pods. For more information about the `product.xml` file, see [Chapter 11, “The product.xml File,” on page 395](#).

Be aware that there is no limit on the number of pods that can be opened by an application. If your application opens pods in response to user interaction, it is a good idea to include code that checks for the presence of a pod before adding another one.

## Initializing a pod

When you add a pod to the list of pods with the `pod` tag or the `addPod()` method and then display it with the `viewPod()` function, Central launches the specified pod SWF file, and the code in the SWF file begins to execute. To properly initialize the pod, you should include ActionScript that performs the following tasks:

- Defines a class that implements the `mx.central.Pod` interface.
- Instantiates the class.
- Calls `mx.central.Central.initPod()`. This is analogous to the `mx.central.Central.initApplication()` call made in an application.
- Implements the `onActivate()` method to initialize pod variables and start running the pod.

The Hello World sample application in the Central SDK contains a `HelloPod` class (in the file `HelloPod.as`).

The `HelloPod` class and its instance variables are defined in the `HelloPod.as` file as follows:

```
import mx.central.Console;

class HelloPod implements mx.central.Pod {
    var oRoot:MovieClip;
    var oConsole:Console;
    var nPodID:Number;
    var nViewerID:Number;
    var nPosition:Number;
    var nBaseTabIndex:Number;
    var nLastTabIndex:Number;
    var oData:Object;
```

And, the `HelloPod()` constructor method accepts a `MovieClip` which corresponds to the main timeline for the Pod:

```
/**
 * Constructor. Saves a reference to the pod's main movie clip for future use.
 */
function HelloPod(mc:MovieClip) {
    oRoot = mc;
}
```

The Central Console calls the `onActivate()` method in response to the `initPod()` method. In this method you store the input parameters and then begin the main functionality of your pod:

```
/**
 * Called by the Shell when our application is loaded.
 */
function onActivate (console:Console, podID:Number, viewerID:Number,
    position:Number, baseTabIndex:Number, initialData:Object):Void
{
    this.oConsole = console;
    this.nPodID = podID;
    this.nViewerID = podID;
    this.nPosition = position;
    this.nBaseTabIndex = baseTabIndex;
    this.oData = initialData;

    // there is only one tab-able control in this pod
    this.nLastTabIndex = baseTabIndex + 1;
}
```

Once your pod class is defined, you can instantiate it from the main timeline, and then call the `initPod()` method to start running the pod in Central. For example:

```
var pod:HelloPod = new HelloPod(this);
mx.central.Central.initPod( this, pod );
```

As shown, the first parameter passed to `initPod()` represents the main SWF for the pod. The second parameter, set to your `HelloPod` class instance, represents the event handler for all pod events.

Central responds to the `initPod()` method by calling the `onActivate()` method on the event handler instance you specified.

## Controlling pods

There are several operations you can perform on a pod in addition to simply displaying it.

- To remove a pod, use the `removePod()` function.

This function can be called from any application part; for example:

```
shellRef.removePod (podID)
consoleRef.removePod (podID)
agentManagerRef.removePod (podID)
```

An application does not need to remove pods when it is uninstalled. Central is responsible for removing all pods installed by an application when it is removed.

- To get the height that was set for a pod when it was first displayed, use the `getHeight()` function.

This function can be called only from the pod:

```
consoleRef.getHeight()
```

By returning the original height of the pod, this method makes it unnecessary for you to duplicate the height information in the pod's `initialData` structure.

- To get information about all the pods that have been added by your application, use the `getPods()` function.

This function can be called from any application part; for example:

```
shellRef.getPods()
consoleRef.getPods()
agentManagerRef.getPods()
```

This function returns an array of `podData` objects for the pods that this application has added (including pods added on product installation). For more information about the `getPods()` method, see [“Displaying a pod in Central” on page 82](#).

- To get the array of currently visible pods associated with your application, use the `getViewedPods()` function.

This function can be called from any application part; for example:

```
shellRef.getViewedPods()
consoleRef.getViewedPods()
agentManagerRef.getViewedPods()
```

This function returns an array of objects for each of this application's pods that are currently displayed in pod viewers.

Each object contains the following properties:

- `podData` An object containing additional properties:
  - `id` A number indicating the internal ID assigned by Central to the pod. This number is read-only and is assigned by Central rather than by the developer.
  - `name` A string indicating the name of the pod as defined by the `name` attribute of the pod tag.
  - `className` A string indicating the name of the SWF file for the pod as defined by the `src` attribute of the pod tag.
  - `height` A number indicating the height of the pod in pixels as defined by the `height` attribute of the pod tag.
  - `initialData` An optional object containing properties defined by the developer. This object can be defined and then passed to the `addPod()` method. For more information about the `addPod()` method, see the next section.
- `viewerID` A unique identifier string for the viewer that the pod is currently being viewed in. This identifier can be used to create individual local connections or otherwise uniquely identify the pod view.

- **position** A number indicating the position of the pod in the Console. Positions are numbered from the top of the Console to the bottom. This may change over the life of the pod as the user views more pods or removes viewers.
- **collapsed** A Boolean value (`true`, `false`) that indicates whether the pod is currently in the collapsed state. Collapsed pods are still running, but only the title bar is displayed in the Console.

## Implementing the pod API

The SWF file for your pod should contain the following functions for responding to events from the Console. These functions should be written as a part of a class that implements the `mx.central.Pod` interface. For more information about implementing interfaces in Central, see [“Implementing `mx.central.Application`” on page 38](#).

Function name	Description
<code>onActivate()</code>	Called by Central each time a pod is displayed in the application window. Passes a reference to the Console and other initialization information to the application.
<code>onDeactivate()</code>	Called by Central when the pod is closed, or by the user selecting a different pod in the viewer, closing the viewer, closing the Console, or quitting Central.
<code>onNetworkChange()</code>	Called by Central whenever the user changes the online or offline status of Central. Passes this status information to the pod.
<code>onNoticeEvent()</code>	Called by Central when a notice is dismissed.
<code>getLastTabIndex()</code>	Called by Central to determine the last index that the pod uses for tab ordering for accessibility purposes. This function should return the value of the last index that the pod uses in determining tab order.

### The `onActivate()` function

The `onActivate()` function is called in your application in response to its `mx.central.Central.initPod()` call each time the pod is loaded. A pod gets an `onActivate()` message when it is loaded into the Console, and an `onDeactivate()` message when it is removed from the Console, when its viewer is switched to another pod, or when Central is shut down.

The following parameters are passed to the `onActivate()` function:

```
onActivate (console:mx.central.Console, podID:Number, viewerID:Number,
           position:Number, baseTabIndex:Number, initialData:Object):Void
```

- **console** Callback object used to call into the Console.
- **podID** Unique ID for this instance of the pod. The pod uses this value to establish per-instance connections with other parts of the program group (mini-applications, agents, other pods) and to allocate per-instance local shared objects for persistence. Pod IDs persist when the user quits and restarts Central.



- **viewerID** A secondary ID corresponding to the viewer that the pod is in. This value does not change, even if the position of the viewer changes. This value should be used in conjunction with **podID** to create per-actor local connections, but not for shared memory; viewers from later invocations of **Central** will have different **viewerID** values.
- **position** The position of the pod in the Console. This can be useful if you want the pod to behave differently depending on its position in the Console.
- **baseTabIndex** The starting number the pod should use when setting tab indexes for accessibility.
- **initialData** Application-specific startup data passed from the application, agent, or pod that called the **addPod()** method.

The following code, taken from the **StockWatcher** application example, includes initialization instructions that place the arguments passed to the **onActivate()** function into variables:

```
function onActivate(console:mx.central.Console, id:Number, viewerID:Number,
    position:Number, baseTabIndex:Number, initialData:Object):Void
{
    gConsole = console;
    gAppName += viewerID;

    trace("activate id="+id+" viewerID="+viewerID);

    initStorage();
    initKeyboardListeners();

    // establish local connection to agent for data
    gCom = mx.central.data.LCService.createClient( StockLCService, gAppName,
        this, false );

    // restore last search term
    fSymbol.text = gStorage.data.searchFor;
    setTicker(gStorage.data.lastResult);

    onNetworkChange(gConsole.isConnected());
}
```

For more information, see [“Pod.onActivate\(\)” on page 316](#).

## The onDeactivate() function

The Console calls the **onDeactivate()** function when the pod is about to be unloaded. The pod should save any necessary data, close any active network connections it uses, and clear any global variables and **setInterval()** methods that it uses. For more information, see [“Pod.onDeactivate\(\)” on page 317](#).

## The `onNetworkChange()` function

The Console calls the `onNetworkChange()` function when the network status (online or offline) changes, and passes a Boolean value to the function. A value of `true` indicates that Central is in online mode. A value of `false` indicates that Central is in offline mode. Include code in this function that helps your pod respond appropriately to the change in status. For more information, see [“Pod.onNetworkChange\(\)” on page 318](#).

## The `onNoticeEvent()` function

The `onNoticeEvent()` function is called when a notice created by the application or one of its parts is dismissed. Central passes the following parameters to the function:

- `event` A string describing the notice’s method of dismissal.
- `noticeData` An object containing all the properties of the notice.
- `appData` An object containing application-specific data that was included when the notice was created with the `addNotice()` method.

For more information about these parameters, see [Chapter 6, “Creating Notices,” on page 101](#). For more information about this method, see [“Pod.onNoticeEvent\(\)” on page 319](#).

## The `getLastTabIndex()` function

Central calls the `getLastTabIndex()` function to determine the last index used by the pod for ordering its tab-navigable items for accessibility. This function should return the last index used by the pod. The last index can be calculated by adding the number of indexes used by the pod to the `baseTabIndex` passed to the pod in the `onActivate()` function. For more information, see [“Pod.getLastTabIndex\(\)” on page 315](#).

## Communicating between a pod and the Console

Pods communicate with the Central environment through the methods of the Console. A reference to the Console callback object is provided by the `onActivate()` call when your pod is loaded.

Pods can perform the following tasks through the Console object’s methods:

- Manipulate other pods.  
See [“Creating a pod” on page 81](#).
- Manipulate notices.  
See [Chapter 6, “Creating Notices,” on page 101](#).
- Monitor network connection status.  
See [“Using web services” on page 68](#).
- Cache remote server data locally.  
See [“Caching data locally” on page 64](#).

- Open the associated application.

Use the `Console.loadApplication()` method to load the pod's associated application in the application window. A pod should load an application only as the result of user interaction.

For a complete list of the methods of the Console object, see [“Console object” on page 186](#).



# CHAPTER 5

## Creating an Agent

An *agent* is an optional part of an application that can be used as a place to implement logic that is best kept separate from the other parts of an application. This can include code that monitors remote data for changes, code that updates data displayed in the application or pod, and code that needs to run continuously. Agents are a good place for this kind of code, because they run continuously in the background, even when the application is not running in Macromedia Central. Keeping this kind of logic separate from the other parts of an application allows the application and any pods to stay synchronized with one another and display up-to-date data.

Even if an application has no obvious need for this kind of background logic, using an agent can be a good way to ensure that your application's architecture is robust enough to handle any functionality you want to add in the future. Using an agent with a simple application can make it much easier to add a pod later, because the application will already be written to use distributed logic.

For example, a news application can use an agent to check for new news stories, in the background, and download those stories for display in the application and pod. The agent can also raise a notice to the user when stories about a specified subject are received.

An agent is implemented in a Macromedia Flash SWF file, separate from your application SWF file, that has no user interface and runs only in the background. The agent SWF file is never displayed on the screen. It is important to carefully regulate CPU-intensive operations you might want to perform in an agent, because overusing the processor affects the performance of Central and makes your application unpopular.

### Designing an agent

The following are some things to keep in mind when you design an agent:

- Use an agent as a controller for the other parts of an application. This can be a good design for many types of applications.
- Use an agent to centralize remote data and shared object access functions.
- Don't overuse the processor. Because agents run in the background along with the agents of other applications, use care when writing code that taxes the processor.
- Be aware of system resources.

- Don't use `setInterval()` methods with short intervals.
- Don't load large amounts of remote data often. Do this only when needed, such as when the data has changed.
- Let the user configure the extent to which the agent uses the processor, and limit the user to a reasonable range of values for these settings.

## Creating an agent SWF file

An application can have only one agent. If you want to create an agent, build a SWF file for the agent and place the agent's ActionScript in frame 1. The agent should include the same functions as an application or pod, such as the `onActivate()` and `onDeactivate()` functions. Implement these functions as a part of a class that implements the `mx.central.Agent` interface.

Next, add a reference to the agent in your `product.xml` file. You can then start and stop that agent from your application using methods of the `Shell` object.

The following code shows simple `onActivate()` and `onDeactivate()` functions for an agent, as well as a `think()` function that is called by a `setInterval()` method called in the `onActivate()` function. This code should be in the `MyAgent.as` file.

```
class MyAgent implements mx.central.Agent {

    var gAgentMan:mx.central.AgentManager;
    var gThinkInterval;

    function onActivate(agentManager:mx.central.AgentManager, agentID:Number,
        initialData:Object)
    {
        // remember the agent manager to call functions on it
        gAgentMan = agentManager;

        // start thinking every 30 minutes
        gThinkInterval = setInterval(think, 30 * 60 * 1000);
    }

    // implement other mx.central.Agent methods here:
    // ...

    function think()
    {
        if (gAgentMan.isConnected())
        {
            // do some action here, like call a web service
        }
    }
}
```

The following code should be in your agent SWF file:

```
var agent:MyAgent = new MyAgent();
// tell Central that the Agent is ready
mx.central.Central.initAgent(this, agent);
```

After you build the agent, you need to add it to your application's `product.xml` file. You do this by adding an `agent` tag as a subelement of the `application` tag, declaring the `name` attribute of your agent, and specifying the `src` (location) of the agent SWF file. The value of the `src` attribute can be a path that is relative to the `product.xml` file.

The following XML fragment includes an `agent` tag that specifies only the relative path to the SWF file for the agent:

```
<application name="News Finder"
  src="http://www.myCompany.com/NewsFinder/NewsFinder.swf"
  help="http://www.myCompany.com/"
  enabled="true"
  lang="en"
  background="#FFFFFF"
  version="1"
  width="550" height="400">
  <icon src="http://www.myCompany.com/icons/icon_32.swf" size="32"/>
  <agent name="BetaAppAgent" src="agent.swf"/>
</application>
```

## Starting an agent

You can start agents in two ways. You can set an agent to start automatically when its application is installed and each time Central is started. You can also start an agent programmatically from within an application or pod.

To have an agent start automatically when its application is installed and each time Central is started, add the `started="true"` attribute to the `agent` tag in your `product.xml` file, as the following example shows:

```
<agent name="BetaAppAgent" src="agent.swf" started="true"/>
```

This enables the agent to run whenever Central is running, without the need for any special ActionScript to start the agent.

To start your agent programmatically, use the `startAgent()` method. You can call this method on the Shell or Console object, as the following example shows:

```
shellRef.startAgent()
consoleRef.startAgent()
```

Because an application can have only one agent, the `startAgent()` method does not require any parameters. When the method is called, Central starts the agent SWF file specified in the `product.xml` file. The `startAgent()` method returns `true` if the agent starts successfully. If the agent fails to start, the method returns `false`. In addition, the `startAgent()` method returns `false` if the agent is already running.

An agent's *started* state is preserved when Central exits. Agents that were running when the user quits Central are started again when Central restarts, and stopped agents remain stopped the next time Central starts.

To check whether an agent is running, call the `getAgent()` method and check the `started` property of the object that is returned, as the following example shows:

```
// Make sure the agent is running
var agentData = gShell.getAgent();
if(agentData.started == false)
{
    gShell.startAgent();
}
```

Even *started* agents are not actually running if the user has disabled background tasks in the Central preferences. Users can do this by selecting View > Preferences and deselecting the Allow Tasks to Run in the Background option in the Advanced panel. Your program can check for this using `getPreferences()`, as the following example shows:

```
// Check whether agents are allowed to run
var prefs = gShell.getPreferences();
if(prefs.agentsEnabled == false)
{
    // adjust for running without an agent
}
```

For more information about accessing the Central preferences settings, see [“Working with preferences” on page 57](#).

When an agent starts, its `ActionScript` should not assume that the other parts of the application are already running. The application and pod should not assume that the agent is running, either. This is because the load order of the parts of an application is not necessarily the same each time Central starts. To ensure consistent operation, applications, pods, and agents should check that other application parts are running before attempting to communicate with them.

Use the following methods to determine whether a particular part of your application is running:

- To check whether an agent is running from an application or pod, use `getAgent()`.
- To check whether a pod is running from an application or agent, use `getViewedPods()`.
- To check whether an application is running from a pod or agent, use `getViewedApplications()`.

## Stopping an agent

To stop an agent, use the `stopAgent()` command. This command can be called from an application, pod, or agent, as the following examples show:

```
shellRef.stopAgent()
consoleRef.stopAgent()
agentManagerRef.stopAgent()
```

Call this function from an application or pod to stop the agent associated with it. An agent can also call this function to stop itself.

An application does not need to remove agents when it is uninstalled. Central is responsible for removing all agents installed by an application when it is removed.



## Determining the status of an agent

To determine the status of an agent, use the `getAgent()` command. This command can be called from an application or pod, as the following examples show:

```
shellRef.getAgent ()  
consoleRef.getAgent()
```

The function returns an object that contains the following properties:

- `id` A string that indicates the agent's unique ID.
- `name` A string that indicates the agent's name (as specified in the `product.xml` file).
- `src` A string that indicates the agent's `src` path (specified in the `product.xml` file).
- `started` A Boolean value that indicates whether the agent is currently running.
- `initialData` An object containing the agent's initial data. This object can have multiple properties that you specify in the `initialData` attribute of the `agent` tag in the `product.xml` file. For more information, see [“The `onActivate\(\)` function” on page 98](#).

For more information about the `product.xml` file, see [Chapter 11, “The `product.xml` File,” on page 395](#).

It is possible for agents to run slowly when a lot of them are installed or the user's computer is slow. This condition can cause Central to display a dialog box that says, “A script in this movie is causing unresponsive behavior, do you want to quit script execution?” If the users selects Yes, all the agents shut down. When this happens, the agent cannot be restarted until Central is restarted.

## Implementing the agent API

The following table describes the functions that an agent SWF file can contain for responding to events from the Agent Manager. These functions should be written in a class that implements the `mx.central.Agent` interface. For more information about implementing interfaces in Central, see [“Implementing `mx.central.Application`” on page 38](#).

Function name	Description
<code>onActivate()</code>	Initializes the agent. Called by Central each time an agent starts. Passes a reference to the Agent Manager and other initialization information to the agent.
<code>onDeactivate()</code>	Called by Central when the agent shuts down.
<code>onNetworkChange()</code>	Called by Central whenever the user changes the online/offline status of Central. Passes this status information to the agent.
<code>onNoticeEvent()</code>	Called by Central when a notice is dismissed.
<code>onUninstall()</code>	Called by Central when the agent's application is uninstalled.

## The onActivate() function

Central calls the `onActivate(agentManager, id, initialData)` method when the agent is started and the `mx.central.Central.initAgent()` call has been made. The following parameters are passed to the function:

- `agentManager` A reference to the `agentManager` callback object. Store this in a variable for use in calling the methods of the Agent Manager.
- `id` A unique ID for this agent.
- `initialData` Application-specific data you can define in the `product.xml` file to initialize your agent. To define this data, include an `initialData` tag as a child of the `agent` tag in the `product.xml` file. The data is defined by including attributes and values in the `initialData` tag. The attributes can have any name you choose, and the values must be strings. To use a number, pass it first as a string, and then convert it to a number with the `number()` method.

The following XML fragment shows how you can define `initialData` values in the `product.xml` file:

```
<agent name="BetaAppAgent" src="agent.swf">
  <initialData foo="bar" black="white" good="evil" up="down"/>
</agent>
```

This XML code would result in an object being passed to the `onActivate()` function with the following structure:

```
{
  foo: "bar",
  black: "white",
  good: "evil",
  up: "down",
}
```

For more information about the `onActivate()` function, see [“Initializing an application” on page 36](#) and [“Agent.onActivate\(\)” on page 138](#).

## The onDeactivate() function

The Agent Manager calls the `onDeactivate()` function when the agent is to be unloaded, by an explicit `removeAgent()` call, by a user selecting **Disable Agents** in the preferences, or by Central shutting down. This function is a good place to save any needed data, close network connections, and remove any global variables or `setInterval` objects. For more information, see [“Agent.onDeactivate\(\)” on page 139](#).

## The onNetworkChange() function

The shell calls the `onNetworkChange(connected)` function when the connection status changes. The function is passed a Boolean value that indicates whether Central is in online or offline mode. For more information, see [“Agent.onNetworkChange\(\)” on page 140](#).

## The `onNoticeEvent()` function

Central calls the `onNoticeEvent(event, noticeData, appData)` function when a notice created by the agent's application or pod is dismissed. For more information about the `onNoticeEvent()` function, see [“Agent.onNoticeEvent\(\)” on page 141](#).

## The `onUninstall()` function

Central calls the `onUninstall()` method in an agent when the associated application is uninstalled by the user. This is a good place to delete any shared objects or cached files used by the application. For more information, see [“Agent.onUninstall\(\)” on page 143](#).



# CHAPTER 6

## Creating Notices

Macromedia Central can present notices to the user. Notices are a way of conveying important or timely information to the user in a special area of the Console called the Notice pane. An application, pod, or agent can invoke a notice.

For example, imagine a news application with an agent that checks for stories in the background, even while the news application is not running. The user can configure the application to generate notices when stories about a particular subject appear. The agent checks for new stories about that subject and generates a notice when it finds one. The notice includes a link that the user can click to open the application, which displays the story.

As another example, consider a financial application that generates a notice when the price of a specified stock rises above a user-defined threshold. The notice could simply present the information, or it could provide a link to open the financial application and display the stock's history chart.

Users can choose to disable all notices in Central by clicking the bell icon in the application window.

This chapter describes the API for creating and responding to notices.

### Creating a notice

To create a notice, your application, pod, or agent must implement the Notice API. The following functions are used to work with notices:

- [“The addNotice\(\) function” on page 101](#)
- [“The removeNotice\(\) function” on page 103](#)
- [“The getNotices\(\) function” on page 103](#)

### The addNotice() function

Call this method to create a new notice, as the following examples show:

```
shellRef.addNotice(noticeData, initialData)
consoleRef.addNotice(noticeData, initialData)
agentManagerRef.addNotice(noticeData, initialData)
```

Pass the following two parameters with the method call:

- *noticeData* An object containing properties that describe the content and behavior of the notice. The following are the properties of *noticeData*:
  - name* A string to be displayed in the notice title bar. If not specified by the developer, the default "app name Notice" is used.
  - description* A longer string to be displayed in the body of the notice. The default is an empty string.
  - timeout* An integer that indicates the number of seconds after which the notice is automatically dismissed. To prevent the notice from being dismissed automatically, set the *timeout* property to 0, or leave it undefined.
  - alert* A Boolean value. If set to *true*, indicates to bring the notice to the user's attention, by having the Central icon flash in the task bar (Windows) or bounce in the Dock (Macintosh). If set to *false*, the notice is quietly recorded in the Notice pane of the Console. Users can prevent notices from being brought to their attention by clicking the Mute button in the Notice pane of the Console.
  - engage* A short string to be displayed in the notice as an engage link. Including an engage link in a notice provides a way for the user to invoke an action in the application that generated the notice. If set to *null*, no engage link is displayed. For example, a news application might raise a notice when a new news story is found. The notice can include an engage link that says *Read News*. When the user clicks the link, Central will call the *onNoticeEvent()* method for the news application with the event set to "engage". If the *navigate* property is set to *true*, then the news application also opens and displays the news story.
  - navigate* A Boolean value. If set to *true*, Central switches to the appropriate application if the user clicks the engage link in the notice. If set to *false*, Central does not switch applications, but still calls *onNoticeEvent()*.
- *initialData* An object that contains application-specific data that you define. This data is passed to the *onNoticeEvent()* functions in your application, pods, and agent when the notice is dismissed.

The *addNotice()* method returns a *noticeId* value that you can use to refer to this notice in subsequent function calls.

For example, the following code causes an agent to create a list of new news items and notify the user about them. The notice displays the word *Show* as its engage link, and has its *navigate* property set to *true* so that the application appears when the user clicks the engage link.

```
var notice:Object = new Object();

// set name to show how many new applications there are
notice.name = count+" new application";
if (count > 1)
    notice.name += "s";

// set body description to a list of recent application names
notice.description = "Recently posted";
for (var i=0; i<count; i++)
{
```

```

    notice.description += "\r- "+result.items[i].name;
}

// set engage link name
notice.engageString = "Show";

// specify that the application should open when engaged
notice.navigate = true;

// specify whether notice should be brought to attention
notice.alert = true;

// post notice
noticeID = gConsole.addNotice(notice, {lastcheck: gSearchDate});

```

## The removeNotice() function

Call the `removeNotice()` function to remove a notice from the Notice pane in the Console, as the following examples show:

```

shellRef.removeNotice(noticeID)
consoleRef.removeNotice(noticeID)
agentManagerRef.removeNotice(noticeID)

```

Pass the notice ID that was returned by the `addNotice()` function. In the previous example code, this ID is stored in the `noticeID` variable. When you call the `removeNotice()` method, the `onNoticeEvent()` function triggers and is passed "remove" as the event parameter.

## The getNotices() function

Call the `getNotices()` function to get a list of the currently active notices in the application's program group, as the following examples show:

```

shellRef.getNotices()
consoleRef.getNotices()
agentManagerRef.getNotices()

```

This function returns an array of objects, each containing the `noticeData` and `appData` properties.

## Responding to notices

To respond to notices, each application, pod, and agent should contain an `onNoticeEvent()` function. When the user closes the notice through the engage link or close box, the notice times out, or the application or one of its parts removes the notice programmatically, the `onNoticeEvent()` function is called in each application part that contains it. If your application is not currently open when the user interacts with the notice, you won't receive an event. If the `notice.navigate` property is set to `true`, the application automatically starts when the user clicks the engage link, so you always receive an "engage" event.

Central passes the following parameters to the `onNoticeEvent()` function:

- *event* The notice's method of dismissal. This is an object with a `type` property that contains a string with one of the following four values:
  - "close" The user closed the notice without clicking the notice engage link.
  - "engage" The user dismissed the notice by clicking the engage link.
  - "timeout" The notice was automatically dismissed because of a timeout.
  - "remove" The notice was dismissed by a call to the `removeNotice()` command by the application or one of its parts.
- *noticeData* The object that was included with the original `addNotice()` command. For more information about this object, see [“The addNotice\(\) function” on page 101](#).
- *initialData* The arbitrary data included in the *initialData* parameter of the `addNotice()` command.
- *appID* The ID of the application that generated the notice.

The following code implements an `onNoticeEvent` handler that checks whether the engage link was clicked in the notice, and calls the `Search` method on the `gAppService` web service object:

```
function onNoticeEvent(event, noticeData, initialData, appID)
{
    if (event.type == "engage")
    {
        // if user clicks the engage link in the notice, look for the recent apps
        gAppService.Search({createdDate: initialData.lastcheck});
    }
}
```

## Guidelines for using notices

To provide a pleasant user experience, keep in mind the following guidelines when you implement notices in your applications:

- Invoke notices only when necessary. Try to avoid creating so many notices that they become distracting.
- Provide a way for users to disable notices for your application in your application's custom preferences. Central components include an `IconButton` component. When set to `"icon_alert"`, this component becomes a Set Notices button. You can use this button to open a dialog box (which you can create with the `MDialogBox` component), where you allow the user to set Notices preferences. For more information about implementing preferences for your application, see [“Application-specific preferences” on page 62](#).
- Only set the `alert` property of a notice to `true` when necessary.
- Be careful about setting up notices to automatically open their parent application. Do this only when necessary. Do as little as possible to interrupt the user's normal workflow.



# CHAPTER 7

## Using the Blast Feature

The Blast feature of Macromedia Central provides a way for applications to send user-selected data to other applications installed in Central. Applications can “listen” for this data and respond to it in useful ways.

For example, a user might look up a concert in an entertainment calendar application, and then select the event data in the application. The user can then select a map application from the Blast menu to send the selected data to the map application to get directions. The user might also choose a retail application from the Blast menu to find products related to the performer, such as concert T-shirts.

Another example might be a movie finder application in which the user selects a specific movie. The movie finder application could pass the address of the movie theater to a restaurant finder application to find restaurants near the theater. The restaurant finder could then pass the address of a selected restaurant to a weather application to determine the weather for that location.

The Blast feature can be implemented in an application to enable the application to send data, receive data, or both.

The Blast method of data transfer is based on XML schemas. The use of XML schemas allows for translation between XML and ActionScript objects, and it allows developers to use a common, defined method of describing the data that their applications can provide or consume. To get you started, Macromedia has defined some basic types in the first version of the CentralData.xsd schema.

Although the Blast feature is based on XML schemas, for communication that is based strictly on ActionScript, the schema does not have to exist. For purposes of testing, you can exchange data among your applications by using a schema location and namespace that don't exist. However, if you don't define a schema and make it available, no other developers will be able to interact with your applications.

## Sending data from an application

When the user selects a data item such as a movie listing, restaurant, address, or other item that the developer of the application has defined as something that can be sent to other applications, the application can send the data to other applications by calling the `shellRef.setSelectedItem()` method.

When an application calls the `setSelectedItem()` method, Central displays the Blast menu icon in the user's status bar, indicating that the user has selected a portion of data, such as a movie, restaurant, address, and so on, and displays a string in the status bar with information about the selected item. The Blast menu lists each application that can receive the selected data type, along with its icon. The final item on the Blast menu, All on Screen, enables the user to send the data to all the applications and pods on the screen that can handle the selected data type. If no applications installed in Central handle any of the data types currently selected, the Blast icon is dimmed, and rolling over it displays a tooltip that says, "No other applications can receive this selection."

Control-clicking (Command-clicking on the Macintosh) the Blast menu toggles the Auto Blast feature for the current application. When Auto Blast is enabled, selected items are immediately sent to all applications and pods on the screen that can handle any of the sent data types. The state of the Auto Blast feature is persistent, so if the user turned it on the last time an application was used, it is enabled the next time the application is started. Focus doesn't change during an auto broadcast; it remains with the application that called the `setSelectedItem()` method.

Sending data from an application to other applications requires the following steps:

1. Create an object that describes the data to be sent.
2. Create a `SelectedItem` object that contains a reference to the XML schema that describes the data types you are using.
3. Set a property of the `SelectedItem` object to the object that describes the data.
4. Create an array that contains the `SelectedItem` object(s) that were defined.
5. Call the `shellRef.setSelectedItem()` method.

By implementing these steps, you cause any application that can handle either the address or link data types to appear in the Blast menu when the user clicks it in the Central application window.

### To send data from an application:

1. Create an object that describes the data to be sent. The object must have the properties defined for the data type in the XML schema. For example, the following code defines an address object with the properties defined for the address data type:

```
// create a new object with properties defined by the address
// data type schema.
var tempAddress:Object = new Object();
tempAddress.street = "555 Houston St.";
tempAddress.city = "Berkeley";
tempAddress.state = "California";
tempAddress.postalCode = "94708";
tempAddress.country = "USA";
```

2. Create a `SelectedItem` object that contains a reference to the XML schema that describes the data types you are using. When you construct the `SelectedItem` object, also pass the name of the data type that you are using. The following example uses the address data type:

```
// create a new SelectedItem object so Central knows what the namespace is
var addressSI = new mx.central.data.SelectedItem("http://
download.macromedia.com/pub/central/schemas/CentralData#", "address");
```

3. Set a property of the `SelectedItem` object to the object that describes the data. The property that is set must have the same name as the data type being used, as follows:

```
// set the property of the SelectedItem object with the same name as the XML
// data type (address) to the object previously created with the properties
// defined by the XML schema for the address data type
addressSI.address = tempAddress;
```

4. Create an array that contains the `SelectedItem` object(s) that were defined:

```
// setSelecteditem() expects an array
var tempArray:Array = new Array(addressSI);
```

5. Call the `shellRef.setSelectedItem()` method. The `setSelectedItem()` method takes two arguments. The first is the array defined in the previous step. The second is an optional string to be displayed in the application window status bar.

```
gShell.setSelectedItem(tempArray, "Address selected.");
```

The following `ActionScript` demonstrates the steps required to create a pair of `SelectedItem` objects and pass them to the Central shell with `setSelectedItem()`. It is the complete example from the previous steps. An additional type has been added:

```
// create a new object with properties defined by the address
// data type schema. See the data types definitions for the parts
// of an address object
var tempAddress:Object = new Object();
tempAddress.street = "555 Houston St.";
tempAddress.city = "Berkeley";
tempAddress.state = "California";
tempAddress.postalCode = "94708";
tempAddress.country = "USA";

// create a new SelectedItem object so Central knows what the namespace is
var addressSI = new mx.central.data.SelectedItem("http://
download.macromedia.com/pub/central/schemas/CentralData#", "address");

// set the property of the SelectedItem object with the same name as the XML
// data type (address) to the object you previously created with the properties
// defined by the XML schema for the address data type
addressSI.address = tempAddress;

// repeat the above steps for the <link> data type
var tempLink:Object = new Object();
tempLink.label = "example web page";
tempLink.href = "http://www.macromedia.com";
var linkSI = new mx.central.data.SelectedItem("http://download.macromedia.com/
pub/central/schemas/CentralData#", "link");
linkSI.link = tempLink;
```

```
// setSelectedItem() expects an array
var tempArray:Array = new Array(addressSI, linkSI);

gShell.setSelectedItem(tempArray, "address and web page selected");
```

## Receiving data

The application that the user selects from the Blast menu starts (if it's not already running), and receives an `onActivate()` message normally. After the `onActivate()` function is completed, the application receives an `onSelectedItem()` call from the shell. The `onSelectedItem()` function should contain code that deals with the incoming data in whatever way is appropriate to the application. An application that is “listening” for address data might contain an `onSelectedItem()` function similar to the following one:

```
function onSelectedItem(data)
{
    for (var i:Number = 0; i < data.length; i++)
    {
        // check to be sure the object is an address data type
        if (data[i].address != null)
        {
            // populate an array with the new address object
            addressObjectArray.push(data[i].address);
        }
    }
}
```

**Note:** Even if your application receives only one data type, you should check to be sure that every element of the array is of the right type.

## Sending data from pods

Although pods cannot call `setSelectedItem()` or trigger the Blast menu directly, they can send data to their own application. A pod can use the function `Console.loadApplication()` to open its parent application. You can pass an optional data parameter with `loadApplication()` that triggers the `onSelectedItem()` function in the pod's parent application.

The following `ActionScript` code demonstrates the steps required to create an appropriate `SelectedItem` object and pass it to the parent application with `loadApplication()`. The steps for creating the `SelectedItem` object are the same as for an application.

```
var tempAddress:Object = new Object(); // This will contain an address object.
    See the data types definitions for the parts of an address object
tempAddress.street = "1 Park Avenue";
tempAddress.city = "Oakland";
tempAddress.state = "California";
tempAddress.postalCode = "94708";
tempAddress.country = "USA";

//create a new SelectedItem object so Central knows what the namespace is
var addressSI = new SelectedItem("http://download.macromedia.com/pub/central/
schemas/CentralData#", "address");
```

```
// set the property of the SelectedItem object with the same name as the XML
// data type (address) to the object you previously created with the properties
// defined by the XML schema for the im data type
addressSI.address = tempAddress;

var dataArray:Array = new Array(addressSI);
gConsole.loadApplication(dataArray);
```

Sending data from a pod to its parent application with `loadApplication()` is somewhat less restricted than sending data from one application to another. There is no type checking involved, and you can send data in any format, not just using `SelectedItem` or XML objects. Additionally, you don't have to declare a `supportedType` tag in your `product.xml` file to receive data from your pod.

Pods can receive sent data, but only if the user selects All on Screen from the Blast menu. Like applications, pods receive an `onSelectedItem()` call and must declare the types they support in the `product.xml` file. For more information, see [“Accessing information across domains” on page 77](#).

## Registering supported data types in the product.xml file

Each application and pod can include in its `product.xml` file a list of the data types it can receive. You can register data types for your application with Central by including a `supportedTypes` tag within the `product.xml` `application`, `pod`, or `podClass` tags. When an item is selected in one application, the Blast menu displays only the installed applications that can handle one or more of the types of data selected.

The following `product.xml` example shows a `supportedTypes` tag that describes the namespace, schema, and two supported data types (address and link):

```
<supportedTypes namespace="http://www.myCo.com/CentralData#" schema="http://
  www.myCo.com/CentralData.xsd">

  <type>address</type>
  <type>link</type>

</supportedTypes>
```

When the selected item is broadcast, only active pods and applications that have at least one supported type defined in the `selectedTypes` tag receive the data. Applications that want to receive all data types in a given schema can register to do so by using the XML schema of the type any. For example:

```
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">

  <type>any</type>

</supportedTypes>
```

The namespace attribute is required for the `supportedTypes` tag, but the schema attribute is optional. The effects of having or not having a schema are described in the following sections. Types from multiple schemas can be declared by repeating the `supportedTypes` tag.

## Defining your own data type schema

While it is preferable to use existing data type schemas when possible in order to increase compatibility among applications and limit the number of differing schemas in existence, occasionally it may be necessary to define a new schema for a particular data type.

To define a data type of your own, define the property structure of the data type you want to define. For example, a recipe data type might have properties that describe the recipe name, cuisine type, ingredients, and instructions, as in the following XML code:

```
<recipe>
  <name> string </name>
  <cuisineType> string </cuisineType>
  <ingredients>
    <ingredient type=string amount=string />
  </ingredients>
  <instructions> string </instructions>
</recipe>
```

For additional examples of data type property structures, see [“Data type reference” on page 114](#).

After you have defined the property structure for the data type, you can make a schema definition file that can be published for other developers to refer to. This is also the file that is passed when new `SelectedItem()` is called in `ActionScript`.

You can define any kind of data type you want, but applications created by other developers cannot send or receive the new data type until you publish the type definition file.

## Choosing a schema format

The Blast feature operates by passing `ActionScript` objects between applications. To increase interoperability between applications, the built-in Macromedia data objects are expressed as XML schemas. The following are the three main approaches to using data types in Central:

- Use the existing objects and schemas supplied by Macromedia or other Central developers. For a list of those provided by Macromedia, see [“Data type reference” on page 114](#).
- Get the XML schema that suits your purposes from a schema repository, and create `ActionScript` objects that map to it.
- Create the object definitions and the XML schema yourself.

The first two approaches are preferred because of ease of development and the ability to avoid redundant and incompatible type definitions. If you decide on the third option, helpful information, validators, and tools on writing schemas are on the World Wide Web Consortium XML schema website at [www.w3.org/XML/Schema](http://www.w3.org/XML/Schema).

## Objects with automatic serialization and deserialization

The mapping from XML to `ActionScript` objects is straightforward. The simplest case would be to implement a function such as the following in an application or pod:

```
MyDataInst = function(...)
{
}
```

This creates an ActionScript object called `MyDataInst`. Create an instance to use it in Central, and then attach the data type, by calling the new `SelectedItem()` method, as follows:

```
var sd = new SelectedItem("http://www.myCompany.com/MyData#", "MyDataType")
sd.MyDataType = myDataInst;
```

If you want your application to receive data of this type, list the URL `http://www.myCompany.com/MyData.xsd#MyDataType` in its `product.xml` file. The `MyDataType` property of the `SelectedItem` object is the name of the root element. The notion of an ActionScript object field name mapping to an XML element name is the basis of the automatic serialization or deserialization. The `SelectedItem` object can be seen as analogous to the ActionScript XML object, which acts as a holder for the actual document.

**Note:** If the schema can't be resolved, Central maps the data to the most basic ActionScript-to-XML mapping. However, Macromedia recommends that a schema be provided. This creates some documentation of the data types and allows for better communication with other developers of Central applications.

## Using no schema

When not using a schema, all XML terminals (elements with no children) are deserialized as strings. This means that when serializing an ActionScript object into XML, the generated XML tags do not have any attributes. Using a schema, you can use richer types, like numbers, Boolean values, dates, and so on, and also use XML attributes.

The following is an example of this simple translation.

Start by creating an object called `Foo` with two properties:

```
var fooInst = new Foo("red", "green");

Foo = function(color1, color2)
{
    if(color1) this.color1 = color1;
    if(color2) this.color2 = color2;
}
```

Central translates this object into XML code that looks like the following:

```
<foo>
  <color1>Red</color1>
  <color2>Green</color2>
</foo>
```

However, if a schema had been present and specified that `color1` and `color2` were attributes of the `foo` element, the serialized XML code would look like the following:

```
<foo color1="Red" color2="Green">
</foo>
```

## ActionScript-to-XML type conversion

The following table shows the complete mapping of ActionScript types to XML:

ActionScript type	XML	Description
Object	Element	The name is taken from the name used to address the object in the containing object.
String	Text Node   Attribute Value	The string becomes a text node in the containing element.
Number	Text Node   Attribute Value	The number is converted to a string and becomes a text node in the containing element.
Date	Text Node   Attribute Value	The date is converted to a string and becomes a text node in the containing element.
Boolean	Text Node   Attribute Value	The Boolean value is converted to a string (true or false) and becomes a text node in the containing element.
Array	Element(s)	Each element of the array is serialized, according to the above rules, within an element taken from the name used to address the array in the containing object.

The following table shows the mapping of basic schema types to ActionScript:

Schema type	ActionScript type
xsd:string   xsd:token	String
xsd:decimal   xsd:integer   xsd:negativeInteger   xsd:nonNegativeInteger   xsd:positiveInteger   xsd:nonPositiveInteger   xsd:long   xsd:int   xsd:short   xsd:byte   xsd:unsignedLong   xsd:unsignedInt   xsd:unsignedShort   xsd:unsignedByte   xsd:float   xsd:double	Number
xsd:date   xsd:dateTime   xsd:time	Date
xsd:boolean	Boolean

**Note:** The XML schema serves as a guide for serialization and deserialization, and no validation is done.

## Using XML objects to send data

You can also use ActionScript built-in XML objects to pass data. Again, using the schema is optional, but you have to flag your XML objects with the data type. Taking the preceding example, you might construct the raw XML in the following way:

```
var data:XML = new XML();
var root:XMLNode = x.createElement("foo");
x.appendChild(root);
var bar:XMLNode = x.createElement("bar");
var baz:XMLNode = x.createElement("bar");
root.appendChild(bar);
```



```

root.appendChild(baz);
bar.appendChild(x.createTextNode("Red"));
baz.appendChild(x.createTextNode("Green"));

```

Central would still need to know the data type, so you must attach it to the XML; the namespace is given by the `xmlns` attribute on the root node and the type is given (relative to the namespace) by the `xsi:type` attribute. The following method, provided by Central, does this:

```
data.setType("http://www.myapp.com/Foo#", "Foo");
```

If an application calls the `setSelectedItem()` function with an XML object that has no data type, Central does not know what to do with it and no applications can receive the data.

## Defining your own data type schema

While it is preferable to use existing data type schemas when possible to increase compatibility among applications and limit the number of differing schemas in existence, occasionally it might be necessary to define a new schema for a particular data type.

To define a data type of your own, define the property structure of the data type you want to define. For example, a recipe data type might have properties that describe the recipe name, cuisine type, ingredients, and instructions, as in the following XML code:

```

<recipe>
  <name> string </name>
  <cuisineType> string </cuisineType>
  <ingredients>
    <ingredient type=string amount=string />
  </ingredients>
  <instructions> string </instructions>
</recipe>

```

For additional examples of data type property structures, see [“Data type reference” on page 114](#).

After you have defined the property structure for the data type, you can make a schema definition file that can be published for other developers to refer to. This is also the file that is referenced when a new `SelectedItem` object is created in `ActionScript`.

You can define any kind of data type you want, but applications created by other developers cannot send or receive the new data type until you publish the type definition file.

## Selected item storage

Selected items are not stored by Central. Local shared objects are a good way to store this data if an application requires it. The application is also responsible for clearing the selection by passing a `null` value to the `setSelectedItem()` method if the data becomes irrelevant. Central clears the shell's selected item whenever the shell loads an application.

## Data type reference

Developers are free to use whatever vocabulary they like for their data. Central includes some basic data types for use in Central to help facilitate interoperability across applications.

To use the data types included in Central, reference the schema located at <http://download.macromedia.com/pub/central/schemas/CentralData#> when calling `new SelectedItem()`.

The following example refers to the Macromedia schema definition file, and creates an object based on the “link” data type:

```
linkItem:Object = new Object();
linkItem.label = "Home Page";
linkItem.url = "http://macromedia.com";
newItem = new mx.central.data.SelectedItem("http://download.macromedia.com/
    pub/central/schemas/CentralData#", "link");
newItem.link = linkItem;
```

The following sections describe the Central data types:

### <email>

#### Usage

```
<email type = "string" address = "string" />
```

#### Description

An e-mail address, with an optional type attribute. Common types include "home" and "work".

#### Example

```
<email type = "work" address = "name@domainName.com" />
```

### <phone>

#### Usage

```
<phone type = "string" number="string" />
```

#### Description

A phone number. The type attribute is optional and can contain values like "home", "work", "mobile", or "fax".

#### Examples

```
<phone type = "home" number = "415 555 5445" />
<phone type = "work" number = "+1 415 555 1212" />
<phone type = "mobile" number = "252-1234" />
<phone type = "summer cottage" number = "(217) 555 1212" />
```

## <im>

### Usage

```
<im service = "string" id = "string" kind="string" label="string"/>
```

### Description

An instant messaging address (*id*), with a required *service* attribute and optional *kind* and *label* attributes. The *service* attribute can be one of the following common choices, or another if not listed here:

- AOL
- YIM
- MSN
- ICQ

### Examples

```
<im service = "YIM" ID = "someUserName" />
<im service = "ICQ" ID = "123456789" kind="work" />
```

## <link>

### Usage

```
<link href="string" label = "string" />
```

### Description

A link to a resource on the Internet. The link is contained in the *href* attribute, and there can be an optional *label* attribute that describes the link.

### Examples

```
<link label = "test site" href = "http://www.dookie.com" />
<link label = "Macromedia" href = "http://www.macromedia.com" />
<link href="http://klynch.com" />
```

## <address>

### Usage

```
<address label="string" href="string">
  <street>string</street>
  <city>string</city>
  <state>string</state>
  <province>string</province>
  <region>string</region>
  <postalCode>string</postalCode>
  <country>string</country>
  <other>string</other>
  <full>string</full>
</address>
```

## Description

An address with an optional descriptive `label` attribute, and an optional `href` attribute for providing a link to a directions page. All the child tags of the `address` tag are optional. The `street` and other tags can occur more than once within a single `address` tag. The `full` tag is intended to be used as a string representation of the entire address.

## Examples:

```
<address label="home">
  <street>2331 ward</street>
  <city>berkeley</city>
  <state>ca</state>
  <postalCode>94705-1103</postalCode>
  <country>USA</country>
  <other>c/o the moomen project</other>
</address>

<address label="Macromedia">
  <street>600 Townsend St.</street>
  <city>San Francisco</city>
  <state>CA</state>
  <postalCode>94103</postalCode>
</address>

<address label="Macromedia" href="http://macromedia.com/macromedia/
mmdirections.html">
  <full>600 Townsend St., San Francisco CA 94103</full>
</address>
```

## <contact>

### Usage

```
<contact name="string">
  <email />
  <phone />
  <im />
  <link />
  <address>...</address>
  <coordinates />
</contact>
```

## Description

Contact information for a person. All the items are optional. There can be zero or one `name` attribute, and zero or more of all the other attributes.

## Example

```
<contact name="Jane Doe">
  <email address="jdoe@someCompany.com" />
  <phone type="work" number="123-555-2000" />
  <phone type="home" number="123-555-1212" />
  <im type="work" service="AOL" address="someUser" />
  <link href="http://aCompanyName.com" />
  <address label="home">
```

```

    <street>2331 ward</street>
    <city>berkeley</city>
    <state>ca</state>
    <postalCode>94705-1103</postalCode>
    <country>USA</country>
    <other>c/o the moomen project</other>
  </address>
</contact>

```

## <coordinates>

### Usage

```
<coordinates label="string" latitude = "string" longitude="string" />
```

### Description

A location on earth, expressed in decimal latitude and longitude, along with an optional descriptive label and optional elevation. To convert degrees/minutes/seconds to decimal, use the conversion tool available from the FCC at [www.fcc.gov/](http://www.fcc.gov/).

### Example

```
<coordinates label="Meigs Field, Chicago" latitude = "41.866667" longitude =
  "87.6" />
```

## <business>

### Usage

```

<business domain="string">
  <name>string</name>
  <description>string</description>
  <type>string</type>
  <email />
  <phone />
  <im />
  <link />
  <location country="country-code" postalCode="string" main="yes/no">
    <about />
    <address>...</address>
    <coordinates />
    <serviceRange />
    <hours day="string" open="string" close="string" timezone="string" />
    <parking type="string"> string </parking>
    <publicTransportation type="string" blocksAway="string"> string </
publicTransportation>
    <languageSpoken language="country-code"> string </languageSpoken>
  </location>
</business>

```

### Description

Business description, based on the specification of the SMBmeta (small and medium-size business metadata) format. SMBmeta does not currently contain e-mail, phone, and IM items.

The type of business is specified using the North American Industry Classification System (NAICS). For information about the system, see the Census website at [www.census.gov/epcd/www/naics.html](http://www.census.gov/epcd/www/naics.html). The specific system used is the NAICS 2002, which has codes defined on the 2002 NAICS Codes and Titles page at [www.census.gov/epcd/naics02/naicod02.htm](http://www.census.gov/epcd/naics02/naicod02.htm). You can find the detailed definition of each code by clicking the code numbers on that page, but you may need to check them before choosing a code for your business. For example, codes beginning with 72 are Accommodation and Food Services, 722 are Food Services and Drinking Places, and 722110 are Full Service Restaurants.

The NAICS value is hierarchical, so searches can be done easily on more or less specific depths. When choosing a code for your business, be as specific as possible. If your business falls into more than one category, include all the categories by using multiple `<type>` elements.

You can include specific attributes for a business type by using namespaces, such as Restaurant or Theater, as the following examples show.

### Examples

```
<business>
  <name>Lupa</name>
  <type naics="722110">Restaurant</type>
  <description>The Noe Valley trattoria formerly known as Noi has morphed into
  Lupa, named after the wolf who reared Rome's founders Romulus and Remus;
  owner Stefano Coppola has relaunched it with a Southern Italian bent,
  featuring pastas as well as heartier dishes such as roasted boar on the menu;
  the skinny, bi-level interior, painted lipstick-red and buttery Tuscany
  yellow, has remained intact, but a gas fireplace has been added to the front
  room.</description>
  <phone number="415-642-4664" />
  <location country="us" postalCode="94114">
    <address>
      <street>4109 24th St.</street>
      <other>(bet. Castro & Diamond Sts.)</other>
      <city>San Francisco</city>
      <state>CA</state>
    </address>
    <serviceRange area="neighborhood">Noe Valley</serviceRange>
    <hours open="6:00 PM" close="11:00 PM">Open everyday except holidays</
  hours>
    <restaurant:cuisine type="Italian (Southern)">
  </location>
</business>
<business>
  <name>AMC Kabuki 8</name>
  <type naics="512131">Theater</type>
  <phone number="(415) 922-4AMC" />
  <location country="us" postalCode="94115">
    <address>
      <street>1881 Post Street</street>
      <city>San Francisco</city>
      <state>CA</state>
    </address>
    <theater:feature type="Online Ticketing" />
    <theater:feature type="Handicap Access" />
  </location>
</business>
```

```

    <theater:feature type="Enhanced Sound" />
  </location>
</business>

```

## <publication>

### Usage

```

<publication>
  <title> string </title>
  <creator> string </creator>
  <subject > string </subject >
  <description> string </description>
  <publisher> string </publisher>
  <contributor> string </contributor>
  <date> datetime-string </date>
  <type> string </type>
  <format> string </format>
  <identifier> string </identifier>
  <source> string </source>
  <language> lang-code </language>
  <relation> string </relation>
  <coverage> string </coverage>
  <rights> string </rights>
</publication>

```

### Description

Information about a publication, using the commonly accepted Dublin Core metadata standard. There is a usage guide for this data at <http://dublincore.org/documents/usageguide/>.

### Example

```

<publication>
  <creator>Rose Bush</creator>
  <title>A Guide to Growing Roses</title>
  <description>Describes process for planting and nurturing different kinds of
  rose bushes.</description>
  <date>2001-01-20</date>
  <identifier>ISBN 4535464</identifier>
  <identifier>http://www.amazon.com/exec/obidos/tg/detail/-/1840380713/</
  identifier>
</publication>

```





# CHAPTER 8

## Designing for Central Best Practices

By observing a few guidelines, you can provide an easily understandable user interface, ensure that the Macromedia Central application development process proceeds smoothly, and minimize the size of your application's SWF files. The following sections describe these guidelines.

### Configuring Macromedia Flash

By configuring some settings in the Macromedia Flash authoring environment, you can make the authoring and playback of Central applications proceed smoothly. Use the following configurations for best results:

- Set the frame rate for each Flash file to 21 frames per second. Central runs all applications and pods at this speed, regardless of the actual speed set in the SWF file.
- When publishing Flash files for final deployment (after testing and debugging), use the following settings in the Flash tab of the Publish Settings dialog box:
  - Select Flash Player 7 from the Version menu. Applications run faster when compiled for Flash Player 7. Existing Macromedia Flash Player 6 applications will continue to run as they did in Central 1. If your application must target Flash Player 6, select the “Optimize for FP6, Release 65” checkbox.
  - Select Omit Trace Actions. This prevents `trace()` methods from being executed without requiring them to be manually removed from the file's code.
  - Deselect Debugging Permitted. Central does not support SWD files. This may cause problems during development as well. Use the debugging application supplied with the Central SDK instead.
  - Select Compress Movie. This reduces the size of your SWF files.
- Install one of the Flash authoring templates provided with this SDK. Templates for Flash MX 2004 are located in the Utilities folder. The appropriate template can be installed in Flash using the Macromedia Extension Manager, available at [www.macromedia.com/go/em\\_download](http://www.macromedia.com/go/em_download).

## Application user interface

By applying some simple guidelines to an application's user interface design, you can make the user experience smooth and intuitive. Using the interface components that are built into Central, you can decrease the size of your application's SWF files and greatly simplify the process of building your application. Using these components can also provide a measure of consistency in the user experience across applications from various developers.

Keep in mind the following points when considering the design of your application:

- For detailed information about each component that is built into Central and its APIs, see *Building Central Applications with Components*.
- An MXP file containing the components for authoring is included with this developer's guide. Use the Macromedia Extension Manager to install the MXP into Flash MX 2004.
- Providing a custom help screen for your application can make it easier for users to discover what your application can do and how to access all its features.

To provide help for your application, create an HTML help file and indicate its URL in the `product.xml` `help` tag. This enables the `Help > ApplicationName` menu item in Central. When the user selects this menu item, Central opens the help page in a web browser. The `Help > ApplicationName` menu item uses the `getURL()` method to access the help file. Help files cannot be cached, because the `getURL()` method does not access files in the cache. This means that your help will not be available while in offline mode.

## Central coding conventions

Because applications in Macromedia Central are SWF files loaded into the application shell, which itself is a SWF file, it is important not to write ActionScript that assumes the SWF file is the only SWF file and is at the root level. Follow these rules to create applications that operate smoothly in the Central environment:

- Don't reference the `_level0` identifier.  
References to the `_level0` identifier do not work properly in Macromedia Central, because your application is not the root movie. Also, avoid the slash notation (`/`) sometimes used as an alternative to `_root` for specifying an absolute path to a level. Use relative references such as `_parent` instead.
- Be careful when using the `_root` and `_lockroot` identifiers.  
Central sets `_lockroot` to `true` when a SWF file is loaded. This ensures that when a Central application uses `_root`, it correctly refers to the `_root` of the SWF file. Central applications should not modify the value of `_lockroot`.
- Don't use the `Stage.height` or `Stage.width` property.  
The `Stage` will not be entirely available to your application, because there are surrounding elements in the application window, such as the Launcher bar and status bar. To determine your application's display area, use the `Shell.getBounds()` function.

- Pass global coordinates to `hitTest()`.

Flash developers often pass `_root._xmouse` and `_root._ymouse` to `hitTest()`. In Central, the `_root` for your movie does not necessarily correspond to the global coordinate space. You have to convert the coordinates explicitly:

```
var pt = { x: this._xmouse, y: this._ymouse };
this.localToGlobal( pt );
this.hitTest( pt.x, pt.y );
```

**Note:** Even if you are using `_root`, be sure to call `_root.localToGlobal()`.

- Don't assume that a network connection exists.

Macromedia Central enables your application to run locally even when the user is not online. It's best to design your application so that it functions as much as possible when offline and without assuming that the user will always have a network connection. To determine whether the user is online, call `Shell.isOnline()`. To receive notification of network changes, create an `onNetworkChange` event handler.

- Don't assume that the application is running alone.

Macromedia Central allows users to run multiple applications, each in their own window. Your application can appear in more than one of these windows, all running at the same time. An application should not assume it has only one instance of itself running. For example, when using local connections, be sure to broadcast a unique name for each instance of your application.

- Don't leave `setInterval()` functions running when your application is closed.

Macromedia Central continues running even when your application is deactivated, so threads created with `setInterval()` continue to be called. These tasks should be ended with the `clearInterval()` method when your application receives an `onDeactivate` event.

- Be careful when using `_global` variables. If you choose to use `_global` variables, be sure to delete them in your `onDeactivate` handler. These variables stay in memory after your application has quit, and may interfere with other applications from the same Internet domain. Applications from different Internet domains do not conflict with each other, because Flash maintains a different `_global` space for each domain.

If you choose to use `_global` variables, using namespaces is a good way to help ensure that your variables don't interfere with the variables of other applications. To use namespaces, create objects with names you know to be unique, such as the name of your company combined with the name of the application. Then create variables as properties of those objects.

The following code shows the creation of a global variable in a unique namespace in the `onActivate()` function:

```
function onActivate(shellRef, appID, shellID, baseTabIndex, initialData)
{
    _global.wwwMyCompanyCom.applicationName = new Object();
    _global.wwwMyCompanyCom.applicationName.foo = "bar";
}
```

- For pods and agents, do not use global variables to store instance-specific data.

All the pods from a given domain share the same `_global` object, as do all the agents from a given domain. This makes it easy to share information, but also makes it dangerously easy for different pod or agent instances to overwrite each other's data.

A common mistake in a pod is to store the Console callback object in a global variable, as the following code shows:

```
function onActivate( console, podID, viewerID, position, baseTabIndex,
    initialData )
{
    gConsole = console;
}
```

This is a bad practice. All instances of your pod share the same `gConsole` variable, so it refers to the console callback object for the pod that was created last. Thus, it works correctly only for that pod. In some cases, the callback object does not work at all.

Instead, use ActionScript 2.0 to create a class with all of your Central event methods. Variables referenced by this class are required to be members of the class. This ensures that your class is self-contained and does not pollute the global variable scope.

The following example creates a class to relate the variable to the movie clip:

```
class MyClass implements mx.central.Pod {

    // define gConsole as a class member
    var gConsole:mx.central.Console;

    function onActivate(console:mx.central.Console, podID:Number,
        viewerID:Number, position:Number, baseTabIndex:Number,
        initialData:Object):Void
    {
        gConsole = console;

        [ ... ]
    }

    // other mx.central.Pod methods here
    // ...
}
```

- Use the `onDeactivate()` function to clean up global variables, `setInterval()` functions, and network connections. This also includes any local connections, connections to `LCService`, and any `LCDataProvider` connections.

The following code deletes a global variable in the `onDeactivate()` function:

```
function onDeactivate()
{
    delete _global.wwwMyCompanyCom.foo;
}
```

- Use the `onUninstall()` function to delete shared objects that the application uses. For example, the following function deletes the shared object referred to by the variable `my_so`:  

```
function onUninstall()
{
    my_so.clear()
}
```
- Use fully qualified names to avoid using the same name as another application. For example, `com.mydomain.central.myapplication`.
- Applications and Pods that hide the mouse pointer should detect when the user moves the mouse outside of the application, or pod, area and re-show the mouse pointer appropriately.

## Optimizing SWF files

The following tips can help reduce the size of your application's SWF files:

- Enable the Generate Size Report in Flash option by selecting File > Publish Settings > Flash > Options, and note the information in the report each time you publish the file.
- Remove unused items from the file's library.
- Use the components that are built into Central so you don't have to include components in the SWF files themselves.
- Don't include font definitions in SWF files. Make sure there aren't fonts embedded in text. To verify this, select a portion of text and click the Character button in the Property inspector. Make sure that the No Characters option is selected.
- Turn off compression when determining how changes are affecting the size of the SWF file. The uncompressed savings is stored in RAM when Central is running the application.
- Scan your files for any unused code left over from the development process, and remove it.

## Testing an application

When testing your application in Central, add `trace()` commands to your code and install the Central Debugger application. Central uses `trace()` commands to send information to the Debugger application. By opening the Debugger in a separate application window, you can observe the results of `trace()` commands, as well as additional information, while your application runs.

If an application uses shared objects, the application should be tested with shared objects created by older versions of the application throughout its development.

Be aware of SWD files in Central. To avoid the presence of SWD files, be sure to deselect the Debugging Permitted option in the Flash Publish Settings dialog box for all FLA files used by your application.

## Converting existing Flash applications into Central applications

In general, modifying an existing Flash application for use in Central is a relatively easy task.

The following tasks are required for converting an existing application SWF file to run in Central:

- Add the application APIs so that the application can respond to events from Central. For more information, see [“Implementing the application methods” on page 37](#).
- Modify the ActionScript code to conform to the Central conventions (see [“Central coding conventions” on page 122](#)).
- Add a product.xml file and icons.
- Deploy the application for installation into Central from the web.

# CHAPTER 9

## Deploying Central Applications

After you create an application, you can make it available for download into Macromedia Central. The Software Development Kit license grants you rights to use the SDK for development purposes. You need a deployment license if you want to deploy your application for any purposes other than development testing. Registration is the process of providing information about yourself and details about your application. If you choose to list your application in the Application Finder, you point to the product.xml file that exists with your installation files. When publishing your application, you can provide an Install button on your site, list the application in the Application Finder, or both.

### Deploying an application

To deploy a Central application, follow these steps:

1. Obtain a product ID number from Macromedia. The subsequent tasks require the product ID.
2. Package the application pieces for installation. This includes creating a product.xml file for your application.
3. Post your application for download on a web server.
4. Deploy an installation badge to help users initiate downloading of your application. An installation badge is a SWF file that contains code to start the application download operation.
5. (Optional) Register your application to appear in the Central Application Finder.

### Obtaining a product ID

You must have a product ID to start testing an application in Central. Applications cannot be installed in Central without an ID.

For Central to allow your application to be installed, you must obtain a product ID from Macromedia. You can register your application and obtain the product ID at [www.macromedia.com/go/central\\_productid](http://www.macromedia.com/go/central_productid). The product.xml file requires this ID in order to enable the downloading of your application.

A special product ID is available for use during the testing and development of your application. This product ID can be used by only one application at a time on your local computer, and you must not publish applications for public use with this ID. The development and testing product ID is CND100-062234-167221-651442.

## Packaging the application

To prepare your application for publication and download, create a `product.xml` file for your application. The `product.xml` file provides information that Central uses to locate and install all the necessary files for your application.

After registering your product, update your application's `product.xml` file with the product ID you received during registration, and with the vendor ID you provided during registration. When you publish your application, the Application Finder parses the updated `product.xml` file for details about your application.

List all the elements of your product in the `product.xml` file. You can choose any name you like and place the file wherever you want, as long as you post it to the same domain as your application. Macromedia suggests that you name this file *product.xml* and place it in the same directory as your main application SWF file.

For detailed information about all of the tags in the `product.xml` file, see [Chapter 11, “The product.xml File,” on page 395](#).

The simplest way to package an application is to place all of its related files in a single directory. This includes SWF files for the application, `pod(s)`, `agent`, and `icon(s)`. Any other files the application requires can be placed in this directory as well.

## Posting an application for download

To make an application available for download, upload the directory containing all the application parts to the web server location indicated in the `product.xml` file. This way, when Central parses the `product.xml` file, it will find the application files in the location specified in the `product.xml` file.

## Deploying an installation badge

This SDK includes a ready-made installation badge that can be found in the Utilities folder. An installation badge provides another way to allow users to install applications, in addition to the Application Finder. If you are not planning to list your application in the Application Finder, or if you want to provide users with multiple ways to install applications, use an installation badge.

The installation badge is a SWF file containing ActionScript that performs the following tasks:

- Checks the version of the player installed on the user's computer and prompts the user to update their player if necessary.
- Checks whether Central is installed on the user's computer and installs it if necessary.
- Checks for the `product.xml` file in the same directory where the installation badge SWF file resides. If none is present, Central is installed but the application is not installed into Central.



- Parses the product.xml file and displays the application name, description, and icon within the installation badge itself.
- Installs the application into Central when the user clicks the badge.

Using the installation badge provided by Macromedia makes it very easy to enable users to install your application, and helps provide a consistent installation experience for the user.

You can make your own installation badge as well. To make a SWF file that will install an application into Central, include the ActionScript found in the CentralInstall.as file included in the Utilities folder of the Central SDK, and then call two methods.

The following ActionScript includes the CentralInstall.as file in the current Macromedia Flash MX 2004 file and creates a new fcInstallService object, and then calls the `start()` method of that object:

```
#include "CentralInstall.as"

myIS = new fcInstallService(respObj, url, count);
myIS.start();
```

This code causes your installation SWF file to look for your product.xml file in the same directory where the installation SWF file exists and then install the application based on the information in the product.xml file. Using the CentralInstall.as file gives your installation SWF file all the functionality of the installation badge supplied by Macromedia.

## Publishing an application with the Central Product Setup Wizard

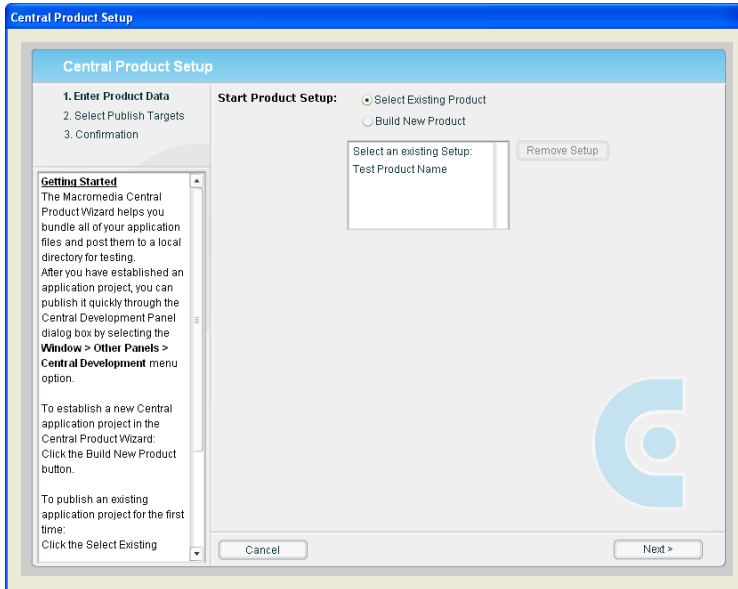
The Central Product Setup Wizard is a developer extension that simplifies the publishing process by organizing and retaining information about your application files. The wizard is installed as part of the authoring extensions included in the Macromedia Central SDK. Use the wizard to organize your application files and establish your product.xml file contents. After your files are categorized and your product.xml file is completed, you can publish subsequent builds of your application with a couple of mouse clicks.

### To organize your application files in the Central Product Setup Wizard

1. Install the authoring extensions that are part of the Macromedia Central SDK.
2. Start the Flash MX 2004 authoring tool (version 7.2 or later).

3. Select the Commands > Central Product Setup menu option.

The Central Product Setup Wizard appears:



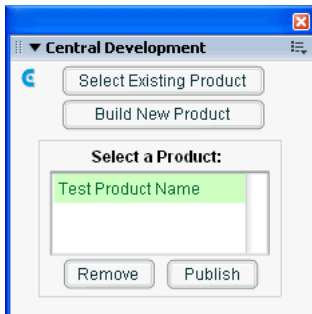
4. Follow the instructions from screen to screen in the wizard to organize your application files and to establish your product.xml file.

After your files are categorized and your product.xml is completed, you can publish subsequent builds of your application files in the Central Development panel.

**To quickly publish a build of your application in the Central Product Setup Wizard:**

1. Select the Window > Other Panels > Central Development menu option.

The Central Development panel appears:



2. In the Central Development panel, click the product name in the list.
3. Click Publish.

## Central and the Application Finder

The Application Finder on the Macromedia website ([www.macromedia.com/go/central\\_appfinder](http://www.macromedia.com/go/central_appfinder)) is the main index of commercial Central applications. Only applications that users must pay for can appear in the Application Finder. The Application Finder is available to users through the Central Products section of the site. A version of the Application Finder runs within Central as well.

You can find all the published applications by logging in to the Macromedia website and going to the Application Finder at [www.macromedia.com/go/central\\_appfinder](http://www.macromedia.com/go/central_appfinder).

In addition to using the information you provide when you register your application, Macromedia Central uses your application's product.xml file to populate the Application Finder with information about your product.



# CHAPTER 10

## API Reference

The Macromedia Central Application Programming Interface (API) consists of methods, properties, and handlers.

### Central API

The Central API can be broken down into the following groups:

**The Environment Management API** The Environment Management API consists of objects that handle calls to and from the Central shell instance running your application.

Object	Description
<a href="#">Central object</a>	Object used to register your application pieces (including pods and agents) as well as provide access to global features such as the LCService object. As a static object, your product always uses the Central object to initialize the application, agent, or pod.
<a href="#">Shell object</a>	Object that manages applications running in the main window. Shell methods are accessed using a reference to the Shell instance received as an argument in the application's <code>onActivate</code> event.
<a href="#">Console object</a>	Object that manages pods running in the Console. Console methods are accessed using a reference to the Console instance received as an argument in the pod's <code>onActivate</code> event.

**The Agent API** The Agent object is an instance of a SWF file (.swf) you build that runs in the background to control flow between different parts of your Central application. This SWF file has no visual state. The AgentManager object manages the agent. The agent is your SWF instance, so it's comparable to an application SWF instance or pod SWF instance. The AgentManager object is equivalent to the Shell or Console object because it manages agent SWF instances just like the Console manages instances of pods, for example.

Object	Description
Agent object	Object containing events made available to your agent instance. Central triggers events that your agent can trap provided that you write callbacks. These callbacks can appear freestanding (that is, with no instance name) as long as you pass the <code>this</code> in the <code>Central.initAgent()</code> call.
AgentManager object	Object that manages agents. Agent Manager methods are accessed using a reference to the AgentManager instance received as an argument in an agent's <code>onActivate</code> event.

**The Application API** The following objects provide the interface for your Central application and help you manipulate application data.

Object	Description
Application object	Object containing events made available to your application SWF instance. Central triggers events that your application can trap provided you write callbacks. These callbacks can appear freestanding (that is, with no instance name) as long as you pass <code>this</code> in the <code>Central.initApplication()</code> call.
SelectedItem object	Object instantiated and filled with data by an application intending to send information to another application or window using the Blast feature.
MD5 object	Object containing a one-way encryption algorithm for checking message integrity.
MovieClip object	Object containing the MovieClip property that identifies a true tooltip when a user's pointer hovers over a particular movie clip.
RegExp object	Object for creating new regular expressions with optional pattern and flags.
String object	String replace method provided as a convenience wrapper for <code>RegExp.replace</code> method.

**The Pod API** The Pod API provides the interface for your Central pods.

Object	Description
Pod object	Object containing events made available to your pod instance. Central triggers events that your pod can trap provided you write callbacks. These callbacks can appear freestanding (that is, with no instance name) as long as you pass <code>this</code> in the <code>Central.initPod()</code> call.

**The Local File Access API** The Local File Access API provides access to files on the user's computer. It also includes capabilities to upload and download.

Object	Description
<a href="#">FileReference object</a>	Object representing a local file. The object can trigger a file browsing dialog box which allows the user to choose a file.
<a href="#">FileReferenceList object</a>	Object representing a group of local files. The object can trigger a file or folder browsing dialog box which allows the user to choose a group of files.

**The Data Service API** LCService and LCDDataProvider are helper classes built into Central Player that make it easy for applications, pods, and agents to communicate using a common channel. The “LC” refers to Local Connection, which is the underlying communication mechanism.

Object	Description
<a href="#">DataProviderClass object</a>	Interface for creating and managing data items.
<a href="#">LCDDataProvider object</a>	An extended version of the DataProviderClass object that allows components in separate parts of your application (application, pod, or agent) to monitor and edit the same set of data.
<a href="#">LCService object</a>	Object instantiated as either the <i>server</i> or a <i>client</i> . Each instance (application, pod, or agent) is given access to methods in the other instances as defined in the object you pass when creating the LCService object.

**Central WebService objects** The following table lists the Central WebService objects. These objects are closely integrated, so when first learning about them, you might want to read the information in the order listed in the table (rather than the alphabetical listing in this document).

Object	Description
<a href="#">WebService object</a>	Constructs a new WebService object for calling web service methods and handling callbacks from the web service. Uses a WSDL file that defines the web service.
<a href="#">PendingCall object</a>	Object returned from a web service method call that you implement to handle the results and faults from that call.
<a href="#">Log object</a>	Optional object used to record activity related to a WebService object.
<a href="#">SOAPCall object</a>	Advanced object that contains information about the web service operation, and provides control over certain behaviors.
<a href="#">RPCFactory object</a>	XML-RPC web service interface that creates calls based on methods defined in the XML-RPC file.
<a href="#">RPC object</a>	XML-RPC web service object that receives events from the RPC method.

## Flash API Deltas

When your Central application is loaded into the Central environment, it is not the top-level SWF file. For this reason, when you are programming applications for Central, a few of the top-level Macromedia Flash APIs should not be used, or have a special-case use. The following table lists the Macromedia Flash APIs, whether they can be used, and a description that includes alternative programming practices.

Element	Use	Description
<code>_root</code>	Yes	The <code>_root</code> property no longer refers to the root of the entire Central application window. In the default case, <code>_root</code> refers to the currently loaded SWF file. Avoid using <code>_root</code> . Use relative paths with <code>_parent</code> instead. This is already a recommendation of Flash API best practices. Many developers simply place the code <code>owner=this</code> at the beginning of their scripts and then use <code>owner</code> as if it were the <code>_root</code> .
<code>_lockroot</code>	Yes	Central manages the <code>_root</code> property with an additional MovieClip property: <code>_lockroot</code> . When your movie clip is loaded, the <code>_lockroot</code> property of your application's SWF file is set to <code>true</code> . When an application refers to <code>_root</code> , Central finds the highest-level MovieClip that has <code>_lockroot</code> set to <code>true</code> . Central applications should not modify the value of <code>_lockroot</code> .
<code>_level0</code>	No	Don't use <code>_level0</code> . Use relative paths with <code>_parent</code> instead. In the Central environment, <code>_level0</code> refers to the top-level Central environment that encompasses all Central applications.
<code>Stage.height</code>	No	<p>Don't rely on <code>Stage.height</code>.</p> <p>To ascertain the current size of your application, call the <code>getBounds()</code> method on the Shell instance. This should return properties for <code>width</code> and <code>height</code>. (If <code>getBounds()</code> returns <code>null</code>, as it will while authoring, you can use <code>Stage.width</code> and <code>Stage.height</code> property.)</p> <p>To refresh your screen when the user resizes, write a handler to trap the <code>onResize()</code> event.</p> <p>To establish the size of your application, provide the width and height as tag attributes of the <code>application</code> tag in your <code>product.xml</code> file.</p> <p>To resize the Central window as your application runs, call <code>Shell.requestSizeChange()</code>. You would do this, for example, if you want your application to be sized at 700 x 700 pixels. If the width and height you pass in a <code>requestSizeChange</code> method are larger than the shell's minimum size, and the minimum size you have specified in your <code>product.xml</code> file, the shell will resize to the parameters you provide in the <code>requestSizeChange</code> method.</p>
<code>Stage.width</code>	No	Don't rely on <code>Stage.width</code> . See the description for <a href="#">Don't rely on Stage.height.</a> , above.



Element	Use	Description
<code>_global</code>	Yes	Variables stored in <code>_global</code> are shared by all applications installed from your domain. Therefore, you can't guarantee that your application is the only one writing to the <code>_global</code> scope. To avoid colliding with your other applications, declare a global object specific to each application, such as <code>_global.myFirstApplication</code> and <code>_global.mySecondApplication</code> , for separate applications running in Central. Finally, make sure to clean up any global variables in your script that trap the events <code>Agent.onDeactivate()</code> , <code>Application.onDeactivate()</code> , or <code>Pod.onDeactivate()</code> .
<code>trace()</code>	Yes	You can use <code>trace()</code> as usual while authoring. Output from your <code>trace()</code> statements can appear in the Central Debug panel for Flash (available in the Central SDK). This way, you can test your application in Central while the output from your <code>trace()</code> statements appears in the Central Debug panel.

## Agent object

**ActionScript Class Name** `mx.central.Agent`

The Agent object is equivalent to your agent SWF instance. That is, your agent SWF instance becomes an instance of the Agent object. The event handlers listed below give your agent a way to react to global events triggered by Central. To use any of these, simply replace *Agent* with *this* (provided you're in your agent SWF instance).

### Method summary for the Agent object

Method	Description
None.	

### Property summary for the Agent object

Property	Description
None.	

### Event handler summary for the Agent object

Event handler	Description
<code>Agent.onActivate()</code>	Called by the AgentManager when the agent is instantiated.
<code>Agent.onDeactivate()</code>	Called by the AgentManager when the agent is to be unloaded.
<code>Agent.onNetworkChange()</code>	Called by the AgentManager when the connection status changes.
<code>Agent.onNoticeEvent()</code>	Called by the AgentManager when a notice created by its application is engaged or closed by the user or gets removed programmatically using a script or a time-out.
<code>Agent.onUninstall()</code>	Called by the shell when your application is being uninstalled from Central.

## Agent.onActivate()

### Availability

Macromedia Central.

### Usage

```
onActivate = function (agentManager, agentID, initialData)
{
    // set a variable to reference the AgentManager
    gManager = agentManager;

    // trigger our own onNetworkChange handler
    // using the current connection status
    this.onNetworkChange(gManager.isConnected());

    // trigger a background task, then repeat every minute
    myBackgroundFunction();
    gMyInterval=setInterval(myBackgroundFunction,60000);
};
```

### Parameters

*agentManager*    agentManager object used this reference to call any functions in the Central Agent API.

*agentID*    Number; unique identifier for this agent.

*initialData*    Any data type; application-specific data. This data is set in the product.xml file's application tag.

### Returns

None.

### Description

Agent callback event; called by the Agent Manager when the agent is instantiated. When your agent SWF instance calls `Central.initAgent()`, this method is called after initialization is complete.

You pass the *initialData* parameter by declaring it in the application's product.xml file using the *initialData* tag. Alternatively, a pod can send *initialData* when it calls *managerReference.loadApplication()*. This way, the data can be dynamic.

A best practice is to keep a reference to the agent manager (*gManager* in this example) so that you have an object onto which you can attach subsequent calls to the Agent Manager API:

```
gManager = agentManager;
if(!gManager.inLocalInternetCache("http://www.mysite.com/my_photo.jpg")){
    gManager.addToLocalInternetCache("http://www.mysite.com/my_photo.jpg");
}
```

In addition, it's good practice to trigger all the handlers that keep your application refreshed once from the `onActivate()` handler. For example, the callback you write for `onNetworkChange()` will be triggered by Central when the connection status changes, but it's a good idea to call it inside the `onActivate()` handler. It's also a good idea to trigger your the `onResize()` event handler to trigger any positioning code so the screen gets organized correctly from the start.

To consolidate the code samples shown for features common to the AgentManager, Shell, and Console objects, many examples show only the first parameter (*agentManager* in this case) being saved in a variable (shown as *gManager*). Although each of these three objects have slightly different implementations of `onActivate`, they all start with a reference to the respective managing object.

## Agent.onDeactivate()

### Availability

Macromedia Central.

### Usage

```
onDeactivate=function()  
{  
    // perform clean up  
    clearInterval(gMyInterval);  
    mySharedObject.data.closingTime=new Date();  
};
```

### Parameters

None.

### Returns

Nothing.

### Description

Agent, application, or pod event handler; called by the respective shell the instant before an agent, application, or pod instance is unloaded. Central triggers the `onDeactivate()` method each time the user uninstalls or updates an application, or exits Central. The `onDeactivate()` event should clean up any global references, including the following:

- Global variables
- Open network connections
- Open Local Connections
- Open LCService and LCDataProvider objects
- Events triggered by `setInterval` (using `clearInterval()`)

Code you place inside the `onDeactivate()` method is ensured to run and is also the last code to execute code before Central shuts down.

## Agent.onNetworkChange()

### Availability

Macromedia Central.

### Usage

```
onNetworkChange = function (connected)
{
    // save connection state in a variable
    gOnline = connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;

    // if online now, try to connect to web services proxy
    if (gOnline==true)
    {
        myBackgroundTask("start");
    }
    else
    {
        myBackgroundTask("stop");
    }
};
```

### Parameters

*connected* Boolean value: true if user is connected; false if offline.

### Returns

Nothing.

### Description

Agent, application, or pod event handler; called by the respective shell when the connection status (online or offline) changes. The shell does not automatically check for connectivity; it simply follows the user setting made in the File menu (either work online or work offline) or when the user selects the network icon (lightning bolt). To determine if the user is online when the application first loads, use the `isConnected()` method. You can then manually trigger your own `onNetworkChange()` handler so that your contained scripts run. That is, Central only triggers `onNetworkChange()` when users manually change their connection status. To check the status using a script, use the `isConnected()` method.

Central can't automatically recognize whether a computer is connected to the Internet; it honors the user's setting.

A best practice is to first check the current status (using `isConnected()`) and save that status in a variable. Do not attempt online access when the status is false. When `onNetworkChange()` reports true (in other words, when going back online), reestablish any background network access, connecting to data, and updating as needed. For example, call a `setInterval` function to periodically call a web service and get up-to-date information. For more information on using the agent to manage data, see [Chapter 2, “Understanding the Macromedia Central Environment,”](#) on page 19.

## Agent.onNoticeEvent()

### Availability

Macromedia Central.

### Usage

```
// Handle a change to an existing Notice from this app
onNoticeEvent = function ( event, noticeData, initialData )
{
    // trace the properties contained in this notice
    trace("event.type="+event.type);
    trace("optional data from issuing app "+initialData);
    for (var i in noticeData){
        trace("noticeData."+i+"="+noticeData[i]);
    }

    // respond according to the event type
    switch (event.type){
        case "close":
            message_txt.text="notice id "+noticeData.id+" was closed";
            break;

        case "engage":
            message_txt.text="you engaged "+noticeData.description;
            break;

        case "timeout":
            message_txt.text="elapsed time reached "+noticeData.timeout;
            break;

        case "remove":
            message_txt.text="removed the notice named "+noticeData.name;
            break;
    }
    // remove this notice from the list of notices we're maintaining
    if(event.type!="engage"){
        myRefreshListOfNotices();
    }
};
```

## Parameters

*event* An object containing one string element, *type*, that provides the reason for the notice's dismissal. The *type* element has one of the following values:

Value	Description
<code>close</code>	Closed by the user by selecting the close box in the notice list.
<code>engage</code>	Closed by the user by selecting the engage button as in the notice detail.
<code>timeout</code>	Dismissed by Central because the notice has timed out.
<code>remove</code>	Dismissed by the application through a call to <code>removeNotice()</code> .

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<code>id</code>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<code>name</code>	A string to be displayed in the notice's title bar. If not specified, the default value of <code>name</code> is: <i>application name</i> Notice
<code>description</code>	Longer string to be displayed in the notice's body. The default is an empty string.
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

*initialData* Any data type specifying application-specific data passed at the time you call `addNotice()`.

## Returns

Nothing.

## Description

Agent, application, or pod event handler; invoked in an agent, application, or pod when a notice created by your application is dismissed. Any of the following events will trigger `onNoticeEvent()`: when the user clicks the close box or clicks the engage text, if the notice times out, or the notice is removed programmatically through the `removeNotice()` method.

The specific values contained in the *noticeData* and *initialData* come from the initial call to `addNotice()`. The `onNoticeEvent()` is not triggered unless your application first creates a notice through `addNotice()`. For an example of how to create a notice and add it, see [AgentManager.addNotice\(\) on page 146](#).

A common use of this method is to include more detail about a notice in a related window. Presumably your user wanted the notice. The `onNoticeEvent()` handler is your opportunity to give the user further details. By passing application-specific data through *initialData*, the application can show the correct item related to a notice (for example, a stock chart view related to a notice about that stock).

## Agent.onUninstall()

### Availability

Macromedia Central.

### Usage

```
onUninstall=function()  
{  
    //execute any final code  
}
```

### Example

```
//this example shows how you can identify your Central customers  
//MAIN WEBSITE SWF:  
mySO=SharedObject.getLocal("centralData", "/");  
if(mySO.data.centralUser==true)  
{  
    message_txt.text="Welcome to my site Central user";  
}  
else  
{  
    message_txt.text="You need to check out my Central app";  
}  
  
//IN YOUR CENTRAL APP:  
onActivate=function(shell)  
{  
    gSO=SharedObject.getLocal("centralData", "/");  
    gSO.data.centralUser=true;  
};  
onUninstall=function()  
{
```

```
gSO.data.centralUser=false;
};
```

### Parameters

None.

### Returns

Nothing.

### Description

Agent and Application event handler; called when the application is being uninstalled from Central. This call gives the application one last chance to clean up (for example, by clearing local shared objects) before being uninstalled. Remember that local shared objects written from Central are stored in the same place as they are when written from your main site. The example shows an application of this fact. When Central itself is uninstalled, this method is not necessarily called for all applications.

## AgentManager object

**ActionScript Class Name** mx.central.AgentManager

Agents communicate with the Central environment through the AgentManager object. The AgentManager is to your agent what the shell is to your application and what the Console is to your pods. That is, there's one AgentManager for all the agents (even though your application can have just one agent). Your agent receives a reference to the AgentManager object as the first parameter in the `onActivate()` method. That reference is used whenever you want to access any methods in the AgentManager object. If you want your agent to communicate directly with your application or pods, you should use your own implementation of the LocalConnection object or the Central LcService class developed specifically for this purpose.

The following methods are implemented by the AgentManager object, and are called by your agent using a reference to the AgentManager. (That is, you'll always replace *AgentManager* with a variable containing the reference to the AgentManager received in your `onActivate()` handler.)

### Method summary for the AgentManager object

Method	Description
<code>AgentManager.addNotice()</code>	Called by an agent to create a new notice.
<code>AgentManager.addPod()</code>	Called by an agent to make a pod available in the console. (The pod doesn't become visible until the user opens it or you call <code>viewPod()</code> .)
<code>AgentManager.addToLocalInternetCache()</code>	Called by an agent to add a URL to the local Internet cache.
<code>AgentManager.inLocalInternetCache()</code>	Returns an array full of ActionScript objects, each containing details about the notices created by your application that are still present.



Method	Description
<code>AgentManager.getPods()</code>	Returns an array full of <code>ActionScript</code> objects—one for each pod available to your application (as listed in the <code>product.xml</code> file or created using <code>addPod()</code> ) and each containing details about that pod.
<code>AgentManager.getPreferences()</code>	Called by an agent to get the user preferences that have been exposed to this application.
<code>AgentManager.getViewedApplications()</code>	Returns an array full of <code>ActionScript</code> objects, each containing details about each <code>Shell</code> instance (that is, separate window) currently running your application.
<code>AgentManager.getViewedPods()</code>	Returns an array full of <code>ActionScript</code> objects, each containing details about the pod instances currently arranged in the Console.
<code>AgentManager.inLocalInternetCache()</code>	Called by an agent when it wants to check if a URL is in the local Internet cache. (Returns <code>true</code> or <code>false</code> .)
<code>AgentManager.isConnected()</code>	Called by an agent to determine current network status. (Returns <code>true</code> or <code>false</code> .)
<code>AgentManager.isConsoleOpen()</code>	Called by an agent to determine if the Console is currently open. (Returns <code>true</code> or <code>false</code> .)
<code>AgentManager.removeFromLocalInternetCache()</code>	Called by an agent to remove a specific URL (such as an image file) from the local Internet cache.
<code>AgentManager.removeNotice()</code>	Called by an agent when it wants to remove a notice using the notice ID returned at the time the notice was added.
<code>AgentManager.removePod()</code>	Called by an agent when it wants to remove a pod using the pod ID returned when the pod was added. (Unlike how a user can close a pod, this makes the pod no longer accessible.)
<code>AgentManager.stopAgent()</code>	Called by an agent if it wants to stop itself.
<code>AgentManager.viewPod()</code>	Calling this function will make the specified pod viewable in the top Viewer (that is, the uppermost tile) of the Console.

## Property summary for the `AgentManager` object

Property	Description
<code>None</code> .	

## Event handler summary for the AgentManager object

Event handler	Description
None.	

### AgentManager.addNotice()

#### Availability

Macromedia Central.

#### Usage

```
noticeID=shellReference.addNotice(noticeData [,initialData])
```

#### Example

```
// This example function adds a notice based on parameters received
// You could use it as follows:
// var thisID=postStockNotice("MACR", 20, "a description", true);
// myListOfNotices.push(thisID);

postStockNotice=function(ticker, price, ruleDescription, alert)
{
    // Creates a new notice object
    var noticeData = new Object();
    noticeData.name = ticker + " " + price;
    noticeData.description = ruleDescription;
    noticeData.alert = alert;
    noticeData.engageString = "show";
    // add noticeData using a reference to gShell (received in onActivate)
    var noticeID = gShell.addNotice(noticeData, {ticker: ticker});

    // return the ID of this notice for future reference
    return noticeID;
}
```

#### Parameters

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<i>id</i>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<i>name</i>	A string to be displayed in the notice's title bar. If not specified, the default value of <i>name</i> is: <i>application name</i> Notice
<i>description</i>	Longer string to be displayed in the notice's body. The default is an empty string.

Property	Description
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

*initialData* Arbitrary application-specific data of any type. This data is received as the third parameter in an `onNoticeEvent` callback.

### Returns

`NoticeID` used to refer to this notice in later calls.

### Description

AgentManager, Console, or Shell method; triggered by an agent, pod, or application, respectively, to create a new notice. You need a reference to the appropriate shell (returned as the first parameter in the `onActivate` event) to which you trigger this method. The examples use `gShell` with the assumption that that variable was set by `onActivate`. For more information on getting a reference to the shell, see `Agent.onActivate`, `Application.onActivate`, or `Pod.onActivate`.

It's good practice to store some identifying information in the optional `initialData` parameter when adding a notice. When the user engages the notice, the identifying information is received in the `onNoticeEvent` event.

Also, it's often better to update a notice instead of adding a new one. You update a notice deleting the old one and replacing it with a new one. This requires you to keep track of the notices as you create them.

## AgentManager.addPod()

### Availability

Macromedia Central.

### Usage

```
podID=shellReference.addPod(podData)
```

### Example

```
// Create a pod when your application loads
```

```

onActivate = function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    var gShell = shell;

    // trigger a homemade function that creates a named pod
    var days=["sun","mon","tue","wed","thu","fri","sat"];
    var dayName=days[new Date().getDay()];
    createPod(dayName+"_pod");
}

// creates and opens a uniquely named pod based on a specific class
createPod = function (theName)
{
    // Create a new pod and populate it
    var podData = new Object();

    // Set the name that displays on the pod itself
    podData.name = theName;

    // This value must be the same as the <podClass name="name"> tag
    podData.className = "calendarClass";

    // Set an initial value to keep with the pod
    podData.initialData = new Date();

    // Add the pod and save a reference to it
    var thisPodID = gShell.addPod( podData );

    // Use the agentManager reference to view this pod in the console
    gShell.viewPod(thisPodID);

};

```

## Parameters

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A *podData* object includes the following properties:

Property	Description
<i>id</i>	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
<i>name</i>	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.

Property	Description
<code>className</code>	<p>A string that specifies the name of the class, as defined in the product.xml file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your <code>pod</code>. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the product.xml file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<code>height</code>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<code>src</code>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the product.xml file's <code>pod</code> tag or <code>podClass</code> tag.)
<code>enabled</code>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<code>appid</code>	A number set by Central to associate a pod with your application.
<code>supportedTypes</code>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the product.xml file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<code>initialData</code>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

## Returns

`podID`; Number set by Central and representing the unique identifier of the pod instance.

## Description

AgentManager, Shell, or Console method; called by an agent, application, or pod, respectively, to add a pod to the Console. The `addPod()` method only makes a new pod instance available, and `viewPod()` actually makes the pod appear (as though the user physically selected it from the Console's pod pop-up menu). You need to use the `podID` returned from the `addPod()` method to trigger the `viewPod()` method.

The hierarchy of application, pod, and podClass is important. As long as your application has at least one pod defined in the product.xml file, the user can instantiate multiple pods in the Console. While the Console only lists applications with pods available, it won't list your application more than once. This is true even if you include multiple pod tags (in the product.xml file) or if you create multiple instances of a pod (either with addPod() or through the product.xml file). If an application has more than one pod available, the user will see that choice in a secondary pop-up menu inside the pod itself (next to where the pod's name appears).

Generally speaking, the user has the ultimate control over how pods are presented. However, through addPod(), your application can make more pods available, and through viewPod() added pods can be displayed. There are also methods to determine which pods are available and which are currently being viewed (getPods() and getViewedPods() respectively). In addition, with a podID you can use the removePod() method to eliminate a particular pod. However, this is not the same as a user closing a pod—removePod() makes the pod unavailable. There are lots of options available, but keep in mind that the goal is to provide the user with intuitive tools that provide flexibility during development.

## AgentManager.addToLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
shellReference.addToLocalInternetCache(url [, bOverwrite, expiration])
```

### Example

```
// this example adds a JPG to the cache, loads it, then checks if successful
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    var theFile="http://www.mysite.com/images/photo.jpg"
    // add it to the cache
    gShell.addToLocalInternetCache(theFile);
    someClipInstance.loadMovie(theFile);

    // check that the disk quota wasn't exceeded
    if(gShell.inLocalInternetCache(theFile)==true)
    {
        // trigger homemade function to explain the image wasn't downloaded
        myAlertFunction("The photo won't be available when offline");
    }

};
```

### Parameters

*url* String; a fully qualified URL where the file to be cached resides.

*bOverwrite* Optional parameter; a Boolean value that indicates whether to overwrite preexisting files of the same name. If the value of *bOverwrite* is `true` and the file indicated as the *url* value is already in the cache, Central overwrites the file. The default value for *bOverwrite* is `false`.

*expiration* Optional parameter; either a `Date` object or a number. This value indicates when the locally cached file will be considered out of date. If you provide a `Date` object for this value, Central considers the file current until the date indicated. If you do not include an expiration date, the default expiration for any cached file is three days. If you provide a number for this value, Central considers the file current for that number of days.

## Returns

None.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to add a URL to the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the cache rather than from the web, enabling products to use data even when the user is offline. To ensure your application loads the URL from the Internet, first call `addToLocalInternetCache()` with the *bOverwrite* parameter set to `true`.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

Usually, you'll call the `addtoInternetCache()` method before loading an image or data file. Regardless of how long the download takes, you can immediately call a command such as `loadMovie()` to load the same file. Adding to the cache simply means that file is saved on the user's hard disk. Although the `addtoIntenetCache()` method does download the file, it primarily adds URLs to a list from which Central always checks before attempting to download from the Internet.

Files do not expire when a user is offline. Similarly, once a file is expired it isn't automatically removed from the cache. Rather, subsequent attempts to load that URL attempt to access the Internet unless the user is offline. If the user is online and the `inLocalInternetCache()` method is called, the file in question will be removed from the cache if it's expired. If the user is online and the `addToInternetCache()` method is called, the file in question will be overwritten if it's expired.

If the value of the *bOverwrite* parameter is `true` and that URL is already in the cache, the file will be overwritten.

The user sets the cache size limit in the Central user preferences. The default size for space shared by all applications running in Central is 20 MB. All applications share this limited space. When the cache contents exceeds 20 MB, the user is asked for more space for local Internet files. If refused, the file is not cached. You can check for success by calling `inLocalInternetCache()`.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

`http://www.mydomain.com/pub/myFile.swf`  
`http://applications.mydomain.com/pub/myFile.swf`

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

**Note:** The size limitation for a URL to add to cache is 129 characters (URLs with more than 129 characters will not be added to cache).

## AgentManager.getNotices()

### Availability

Macromedia Central.

### Usage

```
arrayOfStructures=shellReference.getNotices()
```

### Example

```
// This example creates notice when the user starts or stops your app.
// It uses getNotices() so that it can remove any matching notices.
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    makeNonDuplicatedNotice("STARTUP");
};

onDeactivate=function()
{
    makeNonDuplicatedNotice("SHUTDOWN");
};

makeNonDuplicatedNotice=function(theType){
    // first see if there are any existing notices that match this type
    var currentNotices=gShell.getNotices();
    for(var i=0;i<currentNotices.length;i++)
    {
        var thisNotice=currentNotices[i];
        if(thisNotice.appData==theType)
        {
            gShell.removeNotice(thisNotice.id);
            break;
        }
    }

    // make a new notice
    var now=new Date();
    var noticeData = new Object();
    var initialData = theType;

    // make part unique
    if(theType=="STARTUP")
```



```

    {
        noticeData.name = "Start up time";
        noticeData.description = "You started this app at "+now.toString();
    }
    else
    {
        noticeData.name = "Shut down time";
        noticeData.description = "You closed this app at "+now.toString();
    }

    // set the rest of the properties
    noticeData.alert = false;
    noticeData.navigate = false;
    noticeDataengageString = null;
    noticeData.timeout = 0;

    gShell.addNotice(noticeData, initialData);
};

```

## Parameters

None.

## Returns

An array of structures, each with the following properties:

Property	Description
creationTime	Date object containing the exact time the notice was created.
appID	A unique numeric ID for the application that created the notice. This is the same value received by the <code>onActivate()</code> event.
id	A unique numeric ID for this notice. This is the same value returned when you call <code>addNotice()</code> .
initialData	Can be any data type, passed as the second parameter when you issue <code>addNotice()</code> . For details on how this lets you pack a notice with custom data, see <code>addNotice()</code> .
noticeData	An object that contains general information about the notice, as described next.

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
id	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
name	A string to be displayed in the notice's title bar. If not specified, the default value of <code>name</code> is: <i>application name</i> Notice

Property	Description
<code>description</code>	Longer string to be displayed in the notice's body. The default is an empty string.
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

`AgentManager`, `Console`, or `Shell` method; called by an agent, pod, or application, respectively, to get the currently active notices that your application created. The `getNotices()` method is one of many methods that help you manage the notices you produce. You don't want to inundate your users with useless notices.

When you invoke the `addNotice()` method, an ID number is returned (that you can use when call `removeNotice()`). When a notice is dismissed, the `onNoticeEvent()` callback triggers with complete details. Finally, you can always find complete details regarding the existing notices any time by using the `getNotices()` method.

## AgentManager.getPods()

### Availability

Macromedia Central.

### Usage

```
podData=shellReference.getPods()
```

### Example

```
// displays a list of all available pods and adds an option to remove
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    refreshListOfAllPods();

    myButton.onPress=function(){
        gShell.removePod(myListComponent.getSelected().data);
        refreshListOfAllPods();
    }
}
```

```

    });
};
refreshListofAllPods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of all the pods
    var allPods=gShell.getPods();

    // loop through the pods extracting their names and IDs
    for (var i=0; i<allPods.length; i++)
    {
        var thisPod=allPods[i];
        // add this pod's name and id to an MListBox component instance
        myListComponent.addItem(thisPod.name, thisPod.id);
    }
};

```

For additional examples of this method, see [AgentManager.removePod\(\)](#).

## Parameters

None.

## Returns

An array of `podData` structures for all pods available to this application. The values for a `podData` structure are set in the `product.xml` file or by a script calling `addPod()` or, in the case of `appId` and `id`, by Central itself. For reference, the entire `podData` object documentation is shown below.

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A `podData` object includes the following properties:

Property	Description
<code>id</code>	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
<code>name</code>	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.

Property	Description
<code>className</code>	<p>A string that specifies the name of the class, as defined in the product.xml file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your <code>pod</code>. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the product.xml file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<code>height</code>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<code>src</code>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the product.xml file's <code>pod</code> tag or <code>podClass</code> tag.)
<code>enabled</code>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<code>appid</code>	A number set by Central to associate a pod with your application.
<code>supportedTypes</code>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the product.xml file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<code>initialData</code>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

AgentManager, Console, or Shell method; gets the list of all pods available to this application. This includes pods listed in the product.xml file as well as any created by the `addPod()` method. To get a list of only those pods currently arranged in the Console, use the `getViewedPods()` method instead.

Realize that your application will only be listed once in the Console's pod selection pop-up menu (provided your application has at least one pod). For an application with more than one pod, the user will see a secondary pop-up menu inside the pod (adjacent to the pod's name). The `getPods()` method returns an array of all the pods that will appear in that secondary pop-up menu.

For more information see `getViewedPods()`, `addPod()`, and `viewPod()`.

## AgentManager.getPreferences()

### Availability

Macromedia Central.

### Usage

```
prefObject=ref.getPreferences()
```

### Example

```
// Displays as customized a message as possible at startup
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // get all the preferences
    var all=gShell.getPreferences();

    // prepare a field to populate
    message_txt.text="";

    // encourage them to enable background tasks, just in case
    if(all.agentsEnabled==false)
    {
        message_txt.text+="Please enable background tasks."+newline;
    }

    // if we can't find first or last name, use a generic message
    if(all.userData.firstName==null || all.userData.lastName==null)
    {
        message_txt.text+="Welcome!"+newline;
    }
    else
    {
        message_txt.text+="Welcome "+all.userData.firstName+" "
            +all.userData.lastName+"."+newline;
    }

    // if the locations value isn't null
    if(all.locations!=null)
    {
        // store the location profile from the appropriate index
        var here=all.locations[all.currentLocationIndex];

        // fashion a personalized message to display
        message_txt.text+="You're probably glad to be "
            +here.label+ " in beautiful "+here.city+".";
    }
};
```

## Returns

`pref0Object` Object containing details from the user's global preference settings. Depending on how much access the user has given to your application, you can find the values for some or all of the following properties.

Element	Description
<i>userData</i>	A structure with three properties: {firstName: xxx, lastName: xxx, email: xxx}
<i>locations</i>	An array of structures, each with the following properties: {label: xxx, address1: xxx, address2: xxx, city: xxx, state: xxx, zipcode: xxx, phone: xxx, country: xxx, latitude: xxx, longitude: xxx,}
<i>currentLocationIndex</i>	An index indicating the currently selected location (within the <code>locations</code> array).
<i>agentsEnabled</i>	A Boolean value that indicates whether agents are enabled.

## Description

AgentManager, Console, or Shell method; called by the pod or application to get the general Central preferences the user has exposed to your application. The value for the `agentsEnabled` property matches the user's setting for whether background tasks are allowed (set in the Advanced Preferences dialog box). This value is always available. In fact, you'll also see values for the `userData`, `locations`, and `currentLocationIndex` properties (based on settings under the Identity & Location Preferences dialog box). However, the values are all `null` by default and won't be available until the user has specifically allowed your application access to this data. It's easiest to visualize these properties and subproperties while viewing the Identity & Location Preferences dialog box.

## AgentManager.getViewedApplications()

### Availability

Macromedia Central.

### Usage

```
arrayOfApplicationRecs=shellReference.getViewedApplications()
```

### Example

```
// this example stops users from launching multiple instances of your app
function onActivate(shell)
{
```

```

gShell=shell;

var activeApps = gShell.getViewedApplications();

if(activeApps.length>1)
{
    myAlertDialog("You're already running this app in another window");
}
};

```

### Parameters

None.

### Returns

An array of `applicationRecs` structures that contain details about each instance of your application. Each `applicationRec` structure has the following two properties:

Property	Description
<code>appId</code>	Number indicating the unique ID for your application. (All instances of your application share this number.)
<code>shellID</code>	Number indicating the unique ID for the particular shell running the application. (You can think of this as an ID for the Central shell.)

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of all the shell instances running your application. Naturally, this method only returns information about your applications. You can use the `getViewedApplications()` method to prevent users from launching multiple instances of your application (as the example shows). Also, if you develop a multi-window application, you can use the IDs gathered to set up unique LocalConnection channels (although it's often simpler to use the [LCService object](#)).

## AgentManager.getViewedPods()

### Availability

Macromedia Central.

### Usage

```
arrayOfPodStructures=shellReference.getViewedPods()
```

### Example

```

// displays a detailed list of currently viewed pods
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our refresh function
    refreshListOfActivePods();
}

```

```

// give a button the ability to trigger our refresh function
myButton.onPress=function()
{
    refreshListOfActivePods();
};
};
refreshListOfActivePods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of the currently viewed pods

    var activePods=gShell.getViewedPods();

    // loop through the pods extracting some data for each one
    for (var i=0; i<activePods.length; i++)
    {
        var thisPod=activePods[i];
        var thisLabel=thisPod.podData.name+
            " (slot: "+thisPod.position+") "+
            ((thisPod.collapsed)?"is not open":"is open");

        myListComponent.addItem(thisLabel);
    }
};

```

## Parameters

None.

## Returns

An array of structures for each pod currently visible. The structures have the following properties:

Property	Description
viewerID	Number indicating a unique ID for the pod instance. This is the same number received as the third parameter in the pod's <code>onActivate()</code> handler.
position	Number indicating the current ordinal position of the pod (not the pixel location). Counting from the top, the uppermost pod is in <code>position 0</code> , then <code>position 1</code> , and so on.
collapsed	A Boolean value that indicates whether the pod is in the collapsed state.
podData	A Pod data object as specified in the <code>product.xml</code> file or defined when you call <code>addPod()</code> . For details on the properties contained in a <code>podData</code> object, see <code>getPods()</code> .



## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of this application's currently *viewed* pods in the console. These are simply pods positioned in the Console (regardless of whether the console happens to be open). To get a list of all initialized pods initialized (that is, pods available to the user), regardless of whether they've been loaded in the Console, see [AgentManager.getPods\(\)](#) on page 154.

To determine whether the console is open, use [AgentManager.isConsoleOpen\(\)](#).

The `getViewedPods()` method only returns pods for your application.

While you can use the `removePod()` method (given the `id` property inside the `podData` property) this won't just close the pod but will actually remove it from the pods available to the user. There is no "close pod" method.

## AgentManager.inLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.inLocalInternetCache(url);
```

### Example

```
// this example function loads images with the ultimate user control
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our homemade function
    loadImage("background.jpg");
}

loadImage=function(theImage, override)
{
    var imagePath="http://www.mysite.com/images/"+theImage;

    // if the image is already present (or they're overriding)
    if(gShell.inLocalInternetCache(imagePath)==true || override==true)
    {
        //add the image and load it into a clip instance
        error_txt.text="Loading "+theImage;
        gShell.addToLocalInternetCache(imagePath);
        clipInstance.loadMovie(imagePath);
    }
    else
    {
        // if they're connected
        if(gShell.isConnected()==true)
```

```

    {
        // get their approval by making the button set override to true
        error_txt.text="Do you want to download "+theImage+"?";
        okay_btn.onPress=function(){ loadImage(theImage,true) };
    }
    else
    {
        // if they're not connected just let them try again
        error_txt.text="Connect then press the okay button";
        okay_btn.onPress=function(){ loadImage(theImage)};
    }
}
};

```

### Parameters

*url* A string; fully qualified path that provides the location of the file to be added to the local Internet cache.

### Returns

A Boolean value; `true` if the URL is in the local Internet cache, `false` if not found.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to check if a URL is in the local Internet cache. There are several strategies that might require this method. Before requiring a user to endure a long download, you can first check if the file is available locally, in which case, you get the user's approval first. Also, if the user has indicated that they're not online and the file is not available locally, you can tell them they need to go online first. Finally, the `inLocalInternetCache()` method provides an indirect way to confirm that attempts to add files to the local Internet cache are successful, as described in the following best practice.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

It's a good idea to always confirm the success of any `addToLocalInternetCache()` call by immediately issuing the `inLocalInternetCache()` method with an appropriate follow-up action if the method returns a value of `false`. This is because if the user's 20 MB cache is exceeded and they don't allow an increase, then `addToLocalInternetCache()` effectively fails.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

```

http://www.mydomain.com/pub/myFile.swf
http://applications.mydomain.com/pub/myFile.swf

```

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

## AgentManager.isConnected()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.isConnected()
```

### Example

```
// this example checks the connection state at startup
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // take current state and trigger onNetworkChange (where our code resides)
    this.onNetworkChange(gShell.isConnected());
};

onNetworkChange=function(connected)
{
    // save the connection state in a variable
    gOnline=connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;
};
```

### Parameters

None.

### Returns

A Boolean value: true if the user is connected, false if offline.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to determine current network status. You should consolidate all your code related to connectivity inside the onNetworkChange() handler. Although Central triggers onNetworkChange() automatically, it only does so when the connection status *changes*. Therefore, you need to use isConnected() initially to bring your application in sync. The example shows how an application can trigger its own onNetworkChange() handler (though usually Central does this). This way, all the code is consolidated in one place. There's no reason to repeatedly check isConnected() from multiple places in your code.

## AgentManager.isConsoleOpen()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.isConsoleOpen()
```

### Example

```
// this example adds a notice in order to open a closed Console at startup
onActivate=function(shell)
{
    // set a variable to reference the AgentManager or Shell
    gShell=shell;

    if(gShell.isConsoleOpen()==false)
    {
        var noticeData = new Object();
        noticeData.alert = true;
        noticeData.name = "Welcome to my app!";
        noticeData.description = "You'll need the Console in this app";
        gShell.addNotice(noticeData);
    }
};
```

### Parameters

None.

### Returns

A Boolean value: `true` if the Console is currently open, `false` if it is closed.

### Description

AgentManager or Shell method; called by an agent or an application to determine if the Console is open. You might want to check whether the Console is open before you add or open new pods. Additionally, because some commands cause the Console to open (for example, adding a notice with its `alert` property set to `true`) you should first check whether the Console is open before deciding your approach.

Unlike most methods available from agents, applications, and pods, the `isConsoleOpen()` method is not available from a pod.

## AgentManager.removeFromLocalInternetCache()

### Availability

Macromedia Central.

## Usage

```
shellReference.removeFromLocalInternetCache(URL)
```

## Example

```
// this example attempts to free up space in the user's cache when appropriate
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // add a new image to the local cache and load it into a clip
    var imagePath="http://www.mysite.com/images/";
    var theImage="photo.jpg";
    gShell.addToLocalInternetCache(imagePath+theImage);
    clipInstance.loadMovie(imagePath+theImage);

    // if the new image isn't present that means the cache is full
    if(gShell.inLocalInternetCache(imagePath+theImage)==false)
    {
        // take a list of previously loaded images (could be dynamic)
        myImageList=["big1.jpg", "big2.jpg", "big3.jpg"];

        // and remove each one
        for(var i=0;i<myImageList.length;i++)
        {
            gShell.removeFromLocalInternetCache(imagePath+myImageList[i]);
        }
    }
};
```

## Parameters

*url* Fully qualified location of the file to be removed from the local Internet cache.

## Returns

Nothing.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a URL from the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the web rather than from the cache.

The file in question must be *in* the local Internet cache for this method to work. That means that previously an application must have issued `addToLocalInternetCache()`. Another way to remove a file from the local Internet cache is by setting an expiration date when invoking `addToLocalInternetCache()`. Finally, the `Shell.addToLocalInternetCache()` method also has an `overwrite` parameter which effectively removes a file by replacing it. For more information, see `addToLocalInternetCache()`.

## AgentManager.removeNotice()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.removeNotice(noticeID)
```

### Example

```
// this example creates 3 notices and removes all 3 when any one is dismissed
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // create an array to store IDs for the notices we create
    gPostedNotes=new Array();
    //create a notice, set its name, and add it
    var noticeData = new Object();
    noticeData.engageString="Remove";
    noticeData.name = "One";
    gPostedNotes.push( gShell.addNotice(noticeData) );
    noticeData.name = "Two";
    gPostedNotes.push( gShell.addNotice(noticeData) )
    noticeData.name = "Three";
    gPostedNotes.push( gShell.addNotice(noticeData) )
};

onNoticeEvent=function(event, noticeData, initialData)
{
    // while gPostedNotes still has items remaining
    while(gPostedNotes.length>0)
    {
        // pop one off and remove it
        var thisNotice=gPostedNotes.pop();
        gShell.removeNotice(thisNotice);
    }
};
```

### Parameters

*noticeID* A number identifying the specific notice you want removed. You can use the number returned when you create a notice using the `addNotice()` method. You can also use an `id` property of any object within the array of notices objects returned from the `getNotices()` method.

### Returns

A Boolean value: `true` if the notice was removed, otherwise `false`.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a notice. You can only remove notices that your application created.

The most direct way to track IDs is to store them as you use `addNotices()`. However, this can be difficult because users can dismiss notices (in which case you'll have to trap the `onNoticeEvent()` event) and they might leave the notices untouched when they quit Central (in which case you'll have to save a `LocalShared` object). Probably the easiest tracking method is to include initial data in the second parameter of your `addNotice()` call. Use `getNotices()` and then step through each item returned looking for a matching `appData` property.

A best practice is to minimize the total number of notices by first deleting old notices and then replacing them with new ones containing up-to-date information. For an example of this practice, see [Shell.getNotices\(\)](#).

## AgentManager.removePod()

### Availability

Macromedia Central.

### Usage

```
shellReference.removePod(id)
```

### Example

```
// product.xml excerpt from within the <application> tag:
<podclass name="myRegularPod" src="pod.swf"/>
<podclass name="mySpecialPod" src="specialpod.swf"/>

<pod name="myDefaultPod" className="myRegularPod" />

// this example temporarily exposes a special pod for the user to open
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // give the user the opportunity to access the special pod class
    gCounter=0;
    my_btn.onPress=function()
    {
        gCounter++;
        if(gCounter==3)
        {
            // add a new pod
            var podData=new Object();
            podData.name="special #3";
            podData.className="mySpecialPod";
            gSpecialPodID=gShell.addPod(podData);
        }
        if(gCounter==4)
```

```

        {
            // remove the special pod
            gShell.removePod(gSpecialPodID);
        }
    }
};

```

### Parameters

*id* A number returned when calling the `addPod()` method to create a pod. You can also use an *id* property of any object within the array of objects returned to the `getPods()` method.

### Returns

Nothing.

### Description

`AgentManager`, `Console`, or `Shell` object method; called by an agent, pod, or application, respectively, to remove a pod. Although this method removes from view any pods currently arranged in the Console, it works differently from the way the user manually closes a pod. In the case of `removePod()`, you aren't removing an individual pod SWF instance, but rather removing a pod instance once associated with your application. Naturally, if a matching pod is present in the Console, it must be closed, but using `removePod()` means the user will no longer be able to add that pod instance manually. There is no "close pod" method.

For more information about pods and pod classes see the entry for [Console.addPod\(\)](#). For more information about the difference between application pods and currently viewed pods, see the entries for [Console.getPods\(\)](#) and [Console.getViewedPods\(\)](#), respectively.

Central automatically removes all pods associated with an application when a user uninstalls the application.

## AgentManager.stopAgent()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.stopAgent()
```

### Example

```

// this example stops and starts the agent as connection status changes
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
};

onNetworkChange=function(connected)
{
    if(connected==false)

```



```

    {
        gShell.stopAgent();
    }
    else
    {
        // this only works from an application or pod (not agent)
        gShell.startAgent();
    }
};

```

### Parameters

None.

### Returns

A Boolean value: `true` if the agent was successfully stopped, otherwise `false`.

### Description

AgentManager, Console, or Shell method; called by an agent itself, a pod, or an application to stop the agent SWF file (listed in the product.xml file). After it stops, the agent won't start again until you call `startAgent()` from an application or pod instance. In fact, agents do not start automatically unless the product.xml file's `agent` tag includes the attribute `started="true"`.

**Note:** Your application can only have one agent.

A best practice is to keep as much code as possible in your agent. In such a case, the `stopAgent()` method has questionable value. However, if your agent is primarily executing background tasks using `setInterval()`, it might be easiest to first clear all the intervals and then simply call `stopAgent()`. Additionally, if you design an application to use the agent only temporarily, then it makes sense to use `stopAgent()` (and `startAgent()`).

**Note:** You don't need to remove agents when the user uninstalls your application; Central does this for you.

## AgentManager.viewPod()

### Availability

Macromedia Central.

### Usage

```
shellReference.viewPod( podID [, bForce] )
```

### Example

```

// This example adds a pod only after the user first runs the app.
// It requires you to define a PodClass named "post_install" in the product.xml

onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

```

```

// check to see if the pod is installed
var foundID=null;
var allPods=gShell.getPods();
for(var i=0;i<allPods.length;i++)
{
    if(allPods[i].className=="post_install")
    {
        foundID=allPods[i].id;
        gPodID=foundID;
        break;
    }
}

// create it if it wasn't found
if(foundID==null)
{
    var podData=new Object();
    podData.name="Post Install Pod";
    podData.className="post_install";
    gPodID=gShell.addPod( podData);
}

// give the user a way to view the pod
podOpen_btn.onPress=function()
{
    // should first use getViewedPods to avoid multiple views of this pod
    //as is, this will view the pod added or found above
    gShell.viewPod(gPodID);
};
};

```

## Parameters

*podID* The number returned when calling `addPod()` to create a pod. You can also use an `id` property of any object within the array of objects returned to the `getPods()` method.

*bForce* Optional parameter that forces the creation of a new pod viewer in the Console. The *bforce* parameter is a Boolean value. If *true* (or omitted), `viewPod()` forces the creation of a new pod viewer in the Console. If *false*, a new pod viewer is created only if the pod referred to by `podID` is not already viewed in the Console (so no duplicate pods appear).

## Returns

Nothing.

## Description

AgentManager, Console, or Shell method; called by an agent, application, or pod, respectively, to open a specific pod instance so that it becomes visible in the top slot of the Console. All of the other visible pods are moved down in the console.

To call `viewPod()` you need a `podID` parameter. This means that you either must have used a script to call `addPod()`, which returns an `id`, or used the `getPods()` method to get an array full of structures (each one including an `id`). Using the `getPods()` method might seem to be haphazard if your application has more than one pod, as you wouldn't know which item in the array was for which pod. However, the data in the array returned from `getPods()` includes other details, including any initial data that you can specify in the `product.xml` file. It is probably most intuitive to simply use `viewPod()` immediately after the creation of a pod by using the `addPod()` method, as the example shows.

A best practice is to only call the `viewPod()` method in response to a direct user action. The user should choose how to best populate the Console.

## Application object

**ActionScript Class Name** `mx.central.Application`

The Application object is equivalent to your application SWF instance. That is, your main application SWF instance becomes an instance of the Application object. The methods listed in this section give your application a way to return information to Central when requested. The event handlers listed give your application a way to react to global events triggered by Central. To use any of these, simply replace *Application* with *this* (provided you're in your application SWF instance).

### Method summary for the Application object

Method	Description
<code>Application.getMinimumSize()</code>	Called by the shell to determine the minimum size for your application. You write this method in your application and return data to the shell.
<code>Application.showPreferences()</code>	Called by the shell, when the user chooses the application preferences menu item. This is your cue to trigger code that displays a custom preference dialog box.

### Property summary for the Application object

Property	Description
<code>None.</code>	

## Event handler summary for the Application object

Event handler	Description
<code>Application.onActivate()</code>	Called by the shell when an application is instantiated (that is, becomes active).
<code>Application.onDeactivate()</code>	Called by the shell when a application is about to be taken off screen. (This includes the instant before Central closes if the user exits.)
<code>Application.onNetworkChange()</code>	Called by the shell when the connection status changes.
<code>Application.onNoticeEvent()</code>	Called by the shell when a notice created by its application is engaged or closed by the user or gets removed programmatically using a script or time out.
<code>Application.onPaymentResult()</code>	Reserved. Interface currently unavailable.
<code>Application.onResize()</code>	Called by the shell when the application window is resized.
<code>Application.onSelectedItem()</code>	Called by the shell any time your application receives Blast data. This includes the user manually selecting your application from the Blast menu or when the Send Automatically item is enabled. In either case, it always begins with the other application making a selection of a data type that matches one listed in your product.xml file's <code>supportedTypes</code> tag.
<code>Application.onUninstall()</code>	Called by the shell when your application is being uninstalled from Central.

## Application.getMinimumSize()

### Availability

Macromedia Central.

### Usage

```
getMinimumSize=function()  
{  
    //ensure that your application is never shown smaller than 550x400  
    return {width:550, height:400};  
}
```

### Parameters

None.

### Returns

An object with two properties representing the minimum width and height at which you want to allow your application to be displayed.

*width* Integer; minimum width at which your application can be displayed. Central won't allow minimum widths less than 500.

*height* Integer; minimum height at which your application can be displayed. Central won't allow minimum heights less than 300.

## Description

Application callback method; called by the shell to get the minimum size of your application. This is not a method that you can call, but rather one that you write so that Central (the shell) can find the information it needs. If you do not provide this callback, the default minimum size of your application will be set to `width:500, height:300`. If a user switches from another, smaller, application to your application, the application window will expand to fit the values provided here.

## Application.onActivate()

### Availability

Macromedia Central.

### Usage

```
onActivate = function(shell, appID, shellID, baseTabIndex, initialData)
{
    //set a variable to reference the Shell
    gShell = shell;

    //trigger our own onNetworkChange handler
    //using the current connection status
    this.onNetworkChange(gShell.isConnected());

    //trigger our own onResize handler for layout purposes
    this.onResize();

    //create a unique name for use with LocalConnection
    gUniqueName=appID+"_"+shellID;

    //set the starting tabIndex so our app can be accessible
    gBaseTab=baseTabIndex;
    username_txt.tabIndex=gBaseTab;
    password_txt.tabIndex=gBaseTab+1;
    continue_btn.tabIndex=gBaseTab+2;
};
```

### Parameters

*shell* Shell object; use this reference to call any functions in the Central Shell API.

*appID* Number; unique ID for this application. This number is common to all pieces in your application (application, pods, and agent). Therefore it's appropriate for use in naming your LocalShared objects.

*shellID* String; a unique ID for the particular shell instance (Central window) in which this application is running. This unique string can help identify a single application instance when using LocalConnection objects.

*setBaseTabIndex* Number; used for accessibility—your application should set the tab indexes on controls such as buttons and text fields starting with this number so it doesn't interfere with surrounding shell controls (such as Central's toolbar).

*initialData* Any data type; passed to your application at launch (primarily from the `product.xml` file's `initialData` tag).

## Returns

Nothing.

## Description

Application event handler; called by the shell when your application instantiates. When your SWF instance calls `Central.initApplication()`, this method is called once initialization is complete.

You pass `initialData` by declaring it in the application's `product.xml` file using the `initialData` tag. Alternatively, a pod can send *initialData* when it calls `shellReference.loadApplication()`. This way the data can be dynamic.

A best practice is to keep a reference to the shell hosting your application (`gShell` in this example) so that you have an object onto which you can attach subsequent calls to the Application API:

```
gShell = shell;
if(!gShell.inLocalInternetCache("http://www.mysite.com/my_photo.jpg")){
    gShell.addToLocalInternetCache("http://www.mysite.com/my_photo.jpg");
}
```

In addition, it's a good practice to trigger all the handlers (that keep your application refreshed) once in the `onActivate()` handler. For example:

```
this.onNetworkChange(gShell.isConnected());
```

**Note:** In order to consolidate the code samples shown for features common to the `AgentManager`, `Shell`, and `Console` many examples show just the first parameter (*shell* in this case) being saved in a variable (shown as `gShell`). Although each of these three objects have slightly different implementations of `onActivate`, they all start with a reference to the respective managing object.

## Application.onDeactivate()

### Availability

Macromedia Central.

### Usage

```
onDeactivate=function()
{
    // perform clean up
    clearInterval(gMyInterval);
    mySharedObject.data.closingTime=new Date();
};
```

### Parameters

None.

### Returns

Nothing.

## Description

Agent, application, or pod event handler; called by the respective shell the instant before an agent, application, or pod instance is unloaded. Central triggers the `onDeactivate()` method each time the user uninstalls or updates an application, or exits Central. The `onDeactivate()` event should clean up any global references, including the following:

- Global variables
- Open network connections
- Open Local Connections
- Open LCService and LCDataProvider objects
- Events triggered by `setInterval` (using `clearInterval()`)

Code you place inside the `onDeactivate()` method is ensured to run and is also the last code to execute code before Central shuts down.

## Application.onNetworkChange()

### Availability

Macromedia Central.

### Usage

```
onNetworkChange = function (connected)
{
    // save connection state in a variable
    gOnline = connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;

    // if online now, try to connect to web services proxy
    if (gOnline==true)
    {
        myBackgroundTask("start");
    }
    else
    {
        myBackgroundTask("stop");
    }
};
```

### Parameters

*connected* Boolean value: true if user is connected; false if offline.

### Returns

Nothing.

## Description

Agent, application, or pod event handler; called by the respective shell when the connection status (online or offline) changes. The shell does not automatically check for connectivity; it simply follows the user setting made in the File menu (either work online or work offline) or when the user selects the network icon (lightning bolt). To determine if the user is online when the application first loads, use the `isConnected()` method. You can then manually trigger your own `onNetworkChange()` handler so that your contained scripts run. That is, Central only triggers `onNetworkChange()` when users manually change their connection status. To check the status using a script, use the `isConnected()` method.

Central can't automatically recognize whether a computer is connected to the Internet; it honors the user's setting.

A best practice is to first check the current status (using `isConnected()`) and save that status in a variable. Do not attempt online access when the status is `false`. When `onNetworkChange()` reports `true` (in other words, when going back online), reestablish any background network access, connecting to data, and updating as needed. For example, call a `setInterval` function to periodically call a web service and get up-to-date information. For more information on using the agent to manage data, see [Chapter 2, "Understanding the Macromedia Central Environment," on page 19](#).

## Application.onNoticeEvent()

### Availability

Macromedia Central.

### Usage

```
// Handle a change to an existing Notice from this app
onNoticeEvent = function ( event, noticeData, initialData )
{
    // trace the properties contained in this notice
    trace("event.type="+event.type);
    trace("optional data from issuing app "+initialData);
    for (var i in noticeData){
        trace("noticeData."+i+"="+noticeData[i]);
    }

    // respond according to the event type
    switch (event.type){
        case "close":
            message_txt.text="notice id "+noticeData.id+" was closed";
            break;

        case "engage":
            message_txt.text="you engaged "+noticeData.description;
            break;

        case "timeout":
```



```

        message_txt.text="elapsed time reached "+noticeData.timeout;
        break;

    case "remove":
        message_txt.text="removed the notice named "+noticeData.name;
        break;
    }
    // remove this notice from the list of notices we're maintaining
    if(event.type!="engage"){
        myRefreshListOfNotices();
    }
};

```

## Parameters

*event* An object containing one string element, *type*, that provides the reason for the notice's dismissal. The *type* element has one of the following values:

Value	Description
close	Closed by the user by selecting the close box in the notice list.
engage	Closed by the user by selecting the engage button as in the notice detail.
timeout	Dismissed by Central because the notice has timed out.
remove	Dismissed by the application through a call to <code>removeNotice()</code> .

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
id	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
name	A string to be displayed in the notice's title bar. If not specified, the default value of <i>name</i> is: <i>application name</i> Notice
description	Longer string to be displayed in the notice's body. The default is an empty string.
timeout	A number that specifies the seconds after which the notice should be automatically dismissed. Set <i>timeout</i> to 0 to create a notice that never times out.
alert	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
engageString	A short string; displayed on the engage button. If not specified, no engage button appears.
navigate	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
unread	A Boolean value that indicates whether the notice has been viewed by the user.

*initialData* Any data type specifying application-specific data passed at the time you call `addNotice()`.

### Returns

Nothing.

### Description

Agent, application, or pod event handler; invoked in an agent, application, or pod when a notice created by your application is dismissed. Any of the following events will trigger `onNoticeEvent()`: when the user clicks the close box or clicks the engage text, if the notice times out, or the notice is removed programmatically through the `removeNotice()` method.

The specific values contained in the *noticeData* and *initialData* come from the initial call to `addNotice()`. The `onNoticeEvent()` is not triggered unless your application first creates a notice through `addNotice()`. For an example of how to create a notice and add it, see [AgentManager.addNotice\(\) on page 146](#).

A common use of this method is to include more detail about a notice in a related window. Presumably your user wanted the notice. The `onNoticeEvent()` handler is your opportunity to give the user further details. By passing application-specific data through *initialData*, the application can show the correct item related to a notice (for example, a stock chart view related to a notice about that stock).

## Application.onPaymentResult()

### Availability

Reserved. Interface currently unavailable.

### Description

Reserved. Interface currently unavailable.

## Application.onResize()

### Availability

Macromedia Central.

### Usage

```
onResize=function()  
{  
    //execute layout scripts as the stage size changes  
};
```

### Example

```
onActivate=function(shell)  
{  
    //set a variable to reference the Shell
```

```

gShell=shell;

//trigger our own onResize to set the initial layout
this.onResize();
}
onResize = function()
{
    //ask shell to check app's min & max size
    var bounds = gShell.getBounds();

    //if the bounds weren't returned, just use the stage size
    if (bounds == null)
    {
        bounds = {width: Stage.width, height: Stage.height};
    }
    // layout application items
    centered_mc._x = bounds.width / 2;
    centered_mc._y = bounds.height / 2;
}

```

### Parameters

None.

### Returns

Nothing.

### Description

Application event handler; called by the shell when the application window is resized. You write the `onResize()` handler to trigger custom layout code for when the window is resized by the user for example. To ascertain the correct width and height, call [Shell.getBounds\(\) on page 349](#).

It's a good practice to trigger `onResize()` manually once in the `onActivate()` callback so that any layout scripts will execute at startup. Generally, it's Central that will be calling your application's `onResize()` method.

## Application.onSelectedItem()

### Availability

Macromedia Central.

### Usage

```

onSelectedItem=function(data)
{
    //process the data received
};

```

### Example

//RECEIVER APPLICATION:

```

//place the following in the product.xml's pod and/or application section
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
    <type>any</type>

```

```

</supportedTypes>

//this method will populate an MListBox component
onSelectedItem=function(data)
{
    //prepare a List component to populate
    myListComponent.removeAll();

    for(var i=0;i<data.length;i++)
    {
        //make sure we treat the data as a selectedItem (not XML)
        var thisItem=data[i].asSelectedItem();
        myListComponent.addItem(thisItem.name, thisItem.description);
    }
};

//SENDER APPLICATION:

//this part of the code shows how an application can prepare data to send
onActivate=function(shell){
    //set a variable to reference the Shell or Console
    gShell=shell;

    send_btn.onPress=function
    {
        //prepare the data as an array of two selectedItem items
        var dataToSend=new Array();

        var item1=new SelectedItem("http://www.mysite.com/ns#", "aType");
        item1.name="name one";
        item1.description="this is the description for item 1";
        dataToSend.push(item1);

        var item2=new SelectedItem("http://www.mysite.com/ns#", "aType");
        item2.name="name two";
        item2.description="this is the description for item 2";
        dataToSend.push(item2);

        //with the items prepared, set the data array and a prompt
        gShell.setSelectedItem([item1, item2], "blast two items!");
    };
};

```

### Parameters

*data* An array of instances of the SelectedItem ActionScript structure or an array of XML objects.

### Returns

Nothing.

## Description

Application and pod event handler; called by the shell when data is arriving in your application. For this to happen, another application has to first make a selection of a data type that your application supports. Then a user can manually select your application from the Blast menu or, when the Auto Blast option is enabled, it triggers immediately. Although there are several steps involved in defining the supported data types (in the product.xml file) and preparing a selection to broadcast (in the sending application), `onSelectedItem()` is where you define how your application responds when others send data to it using the Blast feature.

The selection is local to each shell window, so an application in one shell window can set the selected item without destroying the selected item in another shell window.

**Note:** All pods that can receive Blast data will receive it when the user selects the Edit > Blast > All On Screen menu option. Only applications that can receive data show up by name and are listed in the Edit > Blast menu. However, to receive data, pods must still register types that they support in the product.xml file.

Your application needs to handle the received data as an array full of ActionScript structures or an array of XML data. The following two functions are built into Central to make it easy to treat the received data in the form you prefer. (The application that receives the data may not detect in which form it was sent.)

```
asXML(); // returns XML object
asSelectedItem(); // returns SelectedItem object
```

Regardless of whether the data contains structures or XML, you can turn it into the form you want. Macromedia recommends using the ActionScript structure approach for sending and receiving; this eliminates any extra overhead from converting to or from XML, which is naturally more verbose and thus less efficient. You can also use the `asXML()` function to convert data into its XML form for communicating with external sources, debugging, and so on. Similarly, you can use the `asSelectedItem()` function to convert any old XML into `SelectedItem` object instances. Remember, though, to convert any old XML into a `SelectedItem` object properly; the `schemaType` field must be set in order to send XML data directly. For more information on the Blast feature, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Application.onUninstall()

### Availability

Macromedia Central.

### Usage

```
onUninstall=function()
{
    //execute any final code
}
```

### Example

```
//this example shows how you can identify your Central customers
//MAIN WEBSITE SWF:
```

```

mySO=SharedObject.getLocal("centralData", "/");
if(mySO.data.centralUser==true)
{
    message_txt.text="Welcome to my site Central user";
}
else
{
    message_txt.text="You need to check out my Central app";
}

//IN YOUR CENTRAL APP:
onActivate=function(shell)
{
    gSO=SharedObject.getLocal("centralData", "/");
    gSO.data.centralUser=true;

};
onUninstall=function()
{
    gSO.data.centralUser=false;
};

```

### Parameters

None.

### Returns

Nothing.

### Description

Agent and Application event handler; called when the application is being uninstalled from Central. This call gives the application one last chance to clean up (for example, by clearing local shared objects) before being uninstalled. Remember that local shared objects written from Central are stored in the same place as they are when written from your main site. The example shows an application of this fact. When Central itself is uninstalled, this method is not necessarily called for all applications.

## Application.showPreferences()

### Availability

Macromedia Central.

### Usage

```

showPreferences=function()
{
    //display preferences because the user requested to
};

```

### Example

```

showPreferences = function () {

```

```

//get local shared object and save initial value (in case they cancel)
mySO = SharedObject.getLocal("pref");
var initialVal = mySO.data.pref;

//set up two buttons to set one of two colors
colors=[0xFF0000, 0x0000FF];

backgroundColor.setRGB(colors[initialVal]);
preference_mc._visible = true;

preference_mc.pickColorPref = function (_rb)
{
    mySO.data.pref = _rb.value;
    backgroundColor.setRGB(colors[_rb.getValue()]);
};

preference_mc.red_rb.data = 0;
preference_mc.red_rb.value = (initialVal == 0);
preference_mc.red_rb.onRelease = preference_mc.pickColorPref;

preference_mc.blue_rb.data = 1;
preference_mc.blue_rb.value = (initialVal == 1);
preference_mc.blue_rb.onRelease = preference_mc.pickColorPref;

//set up cancel and okay buttons
preference_mc.cancel_btn.onRelease = function()
{
    mySO.data.pref = initialVal;
    backgroundColor.setRGB(colors[initialVal]);
    preference_mc._visible = false;
};

preference_mc.okay_btn.onRelease = function()
{
    preference_mc._visible = false;
};
};

```

### Parameters

None.

### Returns

Nothing.

### Description

Application method; called by the shell, when the user selects the menu option to view your application's preferences. This is where you write code to display an interface from which the user can set their preferences. That is, Central is informing you that the user selected the menu item; it's up to you to display the preferences.

# Central object

**ActionScript Class Name** mx.central.Central

**Note:** For backwards compatibility, Central continues to support the Central 1.0 usage syntax for Central.initAgent, Central.initApplication, and Central.initPod (without using the current class packaging hierarchy). However, new applications should follow the new usage format (mx.central.Central), or import the class.

The Central object is simply a class that lets you tie your application, pod, or agent into the Central environment. You always need to initialize your SWF instances through one of the three methods listed next. After you call the method, Central triggers the `onActivate()` callback you have written inside your SWF instance. From that point forward, you can perform any of the Central specific features.

## Method summary for the Central object

Method	Description
<code>Central.initAgent()</code>	Initializes agents.
<code>Central.initApplication()</code>	Initializes applications.
<code>Central.initPod()</code>	Initializes pods.

## Property summary for the Central object

Property	Description
None.	

## Event handler summary for the Central object

Event handler	Description
None.	

## Central.initAgent()

### Availability

Macromedia Central Player.

### Usage

```
mx.central.Central.initAgent(agentSWF, callbackObject)
```

### Parameters

*agentSWF* An object reference that indicates the agent SWF instance that the AgentManager should initialize. (Use `this` to initialize the SWF file from which you call `initAgent()`.)

*callbackObject* An object reference that indicates the object that the shell should call when issuing callbacks such as `onActivate()`. (Use `this` if you want to define your callbacks inside the SWF file from which you call `initAgent()`.)



## Returns

Nothing.

## Description

Shell method; informs Central that the agent is loaded and is ready to receive callbacks.

Central calls the `onActivate()` method after receiving this `initAgent()` call. It's best to declare all of your event handlers first, and place `Central.initAgent(this, this)` as the last call in your agent SWF file.

## Central.initApplication()

### Availability

Macromedia Central.

### Usage

```
mx.central.Central.initApplication(applicationSWF, callbackObject)
```

### Parameters

*applicationSWF* An object reference indicating the application SWF instance that the shell should initialize. (Simply use `this` to initialize the SWF instance from which you call `initApplication()`.)

*callbackObject* An object reference indicating the object that the shell should call when issuing callbacks such as `onActivate()`. (Simply use `this` if you want to define your callbacks inside the SWF instance from which you call `initApplication()`.)

## Returns

Nothing.

## Description

Shell method; informs Central that the application is loaded and is ready to receive callbacks.

Central calls `onActivate()` after receiving this call. A best practice is to declare all of your event handlers first, and place `Central.initApplication(this, this)` as the last call in your agent SWF file.

## Central.initPod()

### Availability

Macromedia Central.

### Usage

```
mx.central.Central.initPod(podSWF, callbackObject)
```

### Parameters

*podSWF* An object reference indicating the pod SWF instance that the Console should initialize. (Simply use `this` to initialize the SWF instance from which you call `initPod()`.)

*callbackObject* An object reference indicating the object that the shell should call when issuing callbacks such as `onActivate()`. (Simply use `this` if you want to define your callbacks inside the SWF instance from which you call `initPod()`.)

## Returns

Nothing.

## Description

Shell method; informs Central that the pod is loaded and is ready to receive callbacks.

Central calls `onActivate()` after receiving this call. A best practice is to declare all of your event handlers first, and place `Central.initPod(this, this)` as the last call in your agent SWF file.

# Console object

**ActionScript Class Name** `mx.central.Console`

Pods communicate with the Central environment through the Console object. The Console manages pods. The Console is to your pods what the shell is to your application, and what the AgentManager is to your agent. That is, there's one Console for all the pods (including other applications' pods). Your pod receives a reference to the Console as the first parameter in the `onActivate()` method. That reference is used whenever you want to access any methods in the Console object. If you want your pod to communicate directly with your application or agent, you should use your own implementation of the LocalConnection object or the Central [LCService object](#) developed specifically for this purpose.

The following methods are implemented by the Console, and are called by your pods using a reference to the Console. (That is, you always replace *Console* with a variable containing the reference to the Console received in your `onActivate()` handler.)

## Method summary for the Console object

Method	Description
<code>Console.addNotice()</code>	Called by a pod to create a new notice.
<code>Console.addPod()</code>	Called by a pod to make a pod available in the console.
<code>Console.addToLocalInternetCache()</code>	Called by a pod to add a URL to the local Internet cache.
<code>Console.editLocationDialog()</code>	Called by your pod to open the Edit Location dialog box in the same way as if the user manually selects Edit Locations from the Location pop-up menu in the Identity & Location section of the general preferences. This gives the user the opportunity to make changes to their location settings.
<code>Console.getAgent()</code>	Called by a pod to access various properties of the agent, such as whether it's currently running.
<code>Console.getHeight()</code>	Called by a pod to ascertain its own height (specified when it was created).

Method	Description
<code>Console.getNotices()</code>	Returns an array full of ActionScript objects, each containing details about the notices created by your application that are still present.
<code>Console.getPods()</code>	Returns an array of ActionScript objects. One for each pod available to your application (as listed in the product.xml file or created using <code>addPod()</code> ), and each object contains details about that pod.
<code>Console.getPreferences()</code>	Called by a pod to get the user preferences that have been exposed to this application.
<code>Console.getViewedApplications()</code>	Returns an array of ActionScript objects, each containing details about each shell instance (that is, separate window) currently running your application.
<code>Console.getViewedPods()</code>	Returns an array of ActionScript objects, each containing details about the pod instances currently arranged in the Console.
<code>Console.inLocalInternetCache()</code>	Called by a pod when it wants to check whether a URL is in the local Internet cache. (Returns a value of <code>true</code> or <code>false</code> .)
<code>Console.isConnected()</code>	Called by a pod to determine current network status. (Returns a value of <code>true</code> or <code>false</code> .)
<code>Console.loadApplication()</code>	Called by a pod to launch the parent application in a new window.
<code>Console.newLocationDialog()</code>	Called by your pod to launch the New Location dialog box in the same way as if the user manually selects New Locations from the Location pop-up menu in the Identity & Location section of the general preferences. After the user names the new location, the standard preference dialog box appears. You also have the option to tag specific fields as required, although the user can always cancel the operation.
<code>Console.removeFromLocalInternetCache()</code>	Called by a pod to remove a specific URL (such as an image file) from the local Internet cache.
<code>Console.removeNotice()</code>	Called by a pod when it wants to remove a notice using the notice ID returned at the time the notice was added.
<code>Console.removePod()</code>	Called by a pod when it wants to remove a pod using the pod ID returned when that pod was added. (Unlike the way a user can close a pod, this makes the pod no longer accessible.)
<code>Console.startAgent()</code>	Called by a pod to start the agent associated with this application (in the product.xml file).

Method	Description
<code>Console.stopAgent()</code>	Called by a pod to stop the agent associated with this application (in the product.xml file).
<code>Console.viewPod()</code>	Called to make the specified pod viewable in the top Viewer (that is, the uppermost tile) of the console. (This method requires that the specified pod is first identified in your product.xml file or created using <code>addPod()</code> .)

## Property summary for the Console object

Property	Description
None.	

## Event handler summary for the Console object

Event handler	Description
None.	

## Console.addNotice()

### Availability

Macromedia Central.

### Usage

```
noticeID=shellReference.addNotice(noticeData [,initialData])
```

### Example

```
// This example function adds a notice based on parameters received
// You could use it as follows:
// var thisID=postStockNotice("MACR", 20, "a description", true);
// myListOfNotices.push(thisID);

postStockNotice=function(ticker, price, ruleDescription, alert)
{
    // Creates a new notice object
    var noticeData = new Object();
    noticeData.name = ticker + " " + price;
    noticeData.description = ruleDescription;
    noticeData.alert = alert;
    noticeData.engageString = "show";
    // add noticeData using a reference to gShell (received in onActivate)
    var noticeID = gShell.addNotice(noticeData, {ticker: ticker});

    // return the ID of this notice for future reference
    return noticeID;
}
```

## Parameters

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<code>id</code>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<code>name</code>	A string to be displayed in the notice's title bar. If not specified, the default value of <code>name</code> is: <i>application name</i> Notice
<code>description</code>	Longer string to be displayed in the notice's body. The default is an empty string.
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

*initialData* Arbitrary application-specific data of any type. This data is received as the third parameter in an `onNoticeEvent` callback.

## Returns

`NoticeID` used to refer to this notice in later calls.

## Description

`AgentManager`, `Console`, or `Shell` method; triggered by an agent, pod, or application, respectively, to create a new notice. You need a reference to the appropriate shell (returned as the first parameter in the `onActivate` event) to which you trigger this method. The examples use `gShell` with the assumption that that variable was set by `onActivate`. For more information on getting a reference to the shell, see `Agent.onActivate`, `Application.onActivate`, or `Pod.onActivate`.

It's good practice to store some identifying information in the optional `initialData` parameter when adding a notice. When the user engages the notice, the identifying information is received in the `onNoticeEvent` event.

Also, it's often better to update a notice instead of adding a new one. You update a notice deleting the old one and replacing it with a new one. This requires you to keep track of the notices as you create them.

## Console.addPod()

### Availability

Macromedia Central.

### Usage

```
podID=shellReference.addPod(podData)
```

### Example

```
// Create a pod when your application loads
onActivate = function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    var gShell = shell;

    // trigger a homemade function that creates a named pod
    var days=["sun","mon","tue","wed","thu","fri","sat"];
    var dayName=days[new Date().getDay()];
    createPod(dayName+"_pod");
}

// creates and opens a uniquely named pod based on a specific class
createPod = function (theName)
{
    // Create a new pod and populate it
    var podData = new Object();

    // Set the name that displays on the pod itself
    podData.name = theName;

    // This value must be the same as the <podClass name="name"> tag
    podData.className = "calendarClass";

    // Set an initial value to keep with the pod
    podData.initialData = new Date();

    // Add the pod and save a reference to it
    var thisPodID = gShell.addPod( podData );

    // Use the agentManager reference to view this pod in the console
    gShell.viewPod(thisPodID);
};
```

## Parameters

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A *podData* object includes the following properties:

Property	Description
<i>id</i>	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
<i>name</i>	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.
<i>className</i>	<p>A string that specifies the name of the class, as defined in the <code>product.xml</code> file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your pod. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the <code>product.xml</code> file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<i>height</i>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<i>src</i>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the <code>product.xml</code> file's <code>pod</code> tag or <code>podClass</code> tag.)
<i>enabled</i>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<i>appid</i>	A number set by Central to associate a pod with your application.
<i>supportedTypes</i>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the <code>product.xml</code> file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<i>initialData</i>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

## Returns

`podID`; Number set by Central and representing the unique identifier of the pod instance.

## Description

AgentManager, Shell, or Console method; called by an agent, application, or pod, respectively, to add a pod to the Console. The `addPod()` method only makes a new pod instance available, and `viewPod()` actually makes the pod appear (as though the user physically selected it from the Console's pod pop-up menu). You need to use the `podID` returned from the `addPod()` method to trigger the `viewPod()` method.

The hierarchy of application, pod, and `podClass` is important. As long as your application has at least one pod defined in the `product.xml` file, the user can instantiate multiple pods in the Console. While the Console only lists applications with pods available, it won't list your application more than once. This is true even if you include multiple `pod` tags (in the `product.xml` file) or if you create multiple instances of a pod (either with `addPod()` or through the `product.xml` file). If an application has more than one pod available, the user will see that choice in a secondary pop-up menu inside the pod itself (next to where the pod's name appears).

Generally speaking, the user has the ultimate control over how pods are presented. However, through `addPod()`, your application can make more pods available, and through `viewPod()` added pods can be displayed. There are also methods to determine which pods are available and which are currently being viewed (`getPods()` and `getViewedPods()` respectively). In addition, with a `podID` you can use the `removePod()` method to eliminate a particular pod. However, this is not the same as a user closing a pod—`removePod()` makes the pod unavailable. There are lots of options available, but keep in mind that the goal is to provide the user with intuitive tools that provide flexibility during development.

## Console.addToLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
shellReference.addToLocalInternetCache(url [, bOverwrite, expiration])
```

### Example

```
// this example adds a JPG to the cache, loads it, then checks if successful
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    var theFile="http://www.mysite.com/images/photo.jpg"
    // add it to the cache
    gShell.addToLocalInternetCache(theFile);
    someClipInstance.loadMovie(theFile);

    // check that the disk quota wasn't exceeded
```



```

        if(gShell.inLocalInternetCache(theFile)==true)
        {
            // trigger homemade function to explain the image wasn't downloaded
            myAlertFunction("The photo won't be available when offline");
        }

    };

```

## Parameters

*url* String; a fully qualified URL where the file to be cached resides.

*bOverwrite* Optional parameter; a Boolean value that indicates whether to overwrite preexisting files of the same name. If the value of *bOverwrite* is true and the file indicated as the *url* value is already in the cache, Central overwrites the file. The default value for *bOverwrite* is false.

*expiration* Optional parameter; either a Date object or a number. This value indicates when the locally cached file will be considered out of date. If you provide a Date object for this value, Central considers the file current until the date indicated. If you do not include an expiration date, the default expiration for any cached file is three days. If you provide a number for this value, Central considers the file current for that number of days.

## Returns

None.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to add a URL to the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the cache rather than from the web, enabling products to use data even when the user is offline. To ensure your application loads the URL from the Internet, first call `addToLocalInternetCache()` with the *bOverwrite* parameter set to true.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

Usually, you'll call the `addtoInternetCache()` method before loading an image or data file. Regardless of how long the download takes, you can immediately call a command such as `loadMovie()` to load the same file. Adding to the cache simply means that file is saved on the user's hard disk. Although the `addtoIntenetCache()` method does download the file, it primarily adds URLs to a list from which Central always checks before attempting to download from the Internet.

Files do not expire when a user is offline. Similarly, once a file is expired it isn't automatically removed from the cache. Rather, subsequent attempts to load that URL attempt to access the Internet unless the user is offline. If the user is online and the `inLocalInternetCache()` method is called, the file in question will be removed from the cache if it's expired. If the user is online and the `addtoInternetCache()` method is called, the file in question will be overwritten if it's expired.

If the value of the `bOverwrite` parameter is `true` and that URL is already in the cache, the file will be overwritten.

The user sets the cache size limit in the Central user preferences. The default size for space shared by all applications running in Central is 20 MB. All applications share this limited space. When the cache contents exceeds 20 MB, the user is asked for more space for local Internet files. If refused, the file is not cached. You can check for success by calling `inLocalInternetCache()`.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

```
http://www.mydomain.com/pub/myFile.swf
http://applications.mydomain.com/pub/myFile.swf
```

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

**Note:** The size limitation for a URL to add to cache is 129 characters (URLs with more than 129 characters will not be added to cache).

## Console.editLocationDialog()

### Availability

Macromedia Central.

### Usage

```
shellReference.editLocationDialog()
```

### Example

```
// this example gives the user a button they can use to launch the edit dialog
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // set up the button
    edit_btn.onPress=function()
    {
        gShell.editLocationDialog();
    }
}
```

### Parameters

None.

### Returns

Nothing.

## Description

Shell or Console method; called by your application or pod to open the Edit Location dialog box from the user's preferences, so that they may edit their current list of locations. This action is the same as if the user selects Edit Locations from the Location pop-up menu in the Identity & Location section of the general preferences. The only difference here is that the user never sees the rest of their preferences if they cancel the operation. Using the `editLocationDialog()` method is simply a way to help access this setting by way of your application.

## Console.getAgent()

### Availability

Macromedia Central.

### Usage

```
agentData=shellReference.getAgent()
```

### Example

```
// this example lets the user start an agent if it's not already started
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // set up a button to start agent
    turnOnAgent_btn.onPress=function()
    {
        // use getAgent() to find the started property
        if(gShell.getAgent().started==true)
        {
            prompt_txt.text="Already running";
        }
        else
        {
            // attempt to start agent and report the results
            var result=gShell.startAgent();
            prompt_txt.text="Result: "+(result==true)?"success":"failure";
        }
    }
};
```

### Parameters

None.

## Returns

*agentData* Object; contains the following list of properties.

Element	Description
id	A unique numeric ID for the agent. This is the same value received as the second parameter when Central calls the <code>Agent.onActivate()</code> event handler.
name	A string that specifies the name of the agent, which is the same as the name declared for this agent in the <code>product.xml</code> file.
src	A string that specifies the fully qualified location of the SWF file implementing the agent, which is the same as the location declared for this agent in the <code>product.xml</code> file.
started	A Boolean value that indicates whether this agent has been started (that is, whether it's currently running).

Console or Shell method; called by your pod or application to ascertain various properties of the agent. Calling `getAgent()` returns an object with several properties. Considering that most of these properties are hard-wired in your `product.xml` file, the most useful properties are `started` and `enabled`.

## Console.getHeight()

### Availability

Macromedia Central.

### Usage

```
myCurrentHeight=shellReference.getHeight()
```

### Example

```
// this example positions a graphic at the bottom edge of the pod
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    var height=gShell.getHeight();
    borderBottom_mc._y=height;
};
```

### Parameters

None.

### Returns

An integer that represents the pod height in pixels. The default pod height is 100 pixels.

## Description

Console method; called by a pod to determine its height as specified when it was created. When using both the product.xml file and the `addPod()` method, there are options to specify a pod's height.

## Console.getNotices()

### Availability

Macromedia Central.

### Usage

```
arrayOfStructures=shellReference.getNotices()
```

### Example

```
// This example creates notice when the user starts or stops your app.
// It uses getNotices() so that it can remove any matching notices.
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    makeNonDuplicatedNotice("STARTUP");
};

onDeactivate=function()
{
    makeNonDuplicatedNotice("SHUTDOWN");
};

makeNonDuplicatedNotice=function(theType){
    // first see if there are any existing notices that match this type
    var currentNotices=gShell.getNotices();
    for(var i=0;i<currentNotices.length;i++)
    {
        var thisNotice=currentNotices[i];
        if(thisNotice.appData==theType)
        {
            gShell.removeNotice(thisNotice.id);
            break;
        }
    }

    // make a new notice
    var now=new Date();
    var noticeData = new Object();
    var initialData = theType;

    // make part unique
    if(theType=="STARTUP")
    {
        noticeData.name = "Start up time";
        noticeData.description = "You started this app at "+now.toString();
    }
}
```

```

else
{
    noticeData.name = "Shut down time";
    noticeData.description = "You closed this app at "+now.toString();
}

// set the rest of the properties
noticeData.alert = false;
noticeData.navigate = false;
noticeData.engageString = null;
noticeData.timeout = 0;

gShell.addNotice(noticeData, initialData);
};

```

## Parameters

None.

## Returns

An array of structures, each with the following properties:

Property	Description
<code>creationTime</code>	Date object containing the exact time the notice was created.
<code>appId</code>	A unique numeric ID for the application that created the notice. This is the same value received by the <code>onActivate()</code> event.
<code>id</code>	A unique numeric ID for this notice. This is the same value returned when you call <code>addNotice()</code> .
<code>initialData</code>	Can be any data type, passed as the second parameter when you issue <code>addNotice()</code> . For details on how this lets you pack a notice with custom data, see <code>addNotice()</code> .
<code>noticeData</code>	An object that contains general information about the notice, as described next.

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<code>id</code>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<code>name</code>	A string to be displayed in the notice's title bar. If not specified, the default value of <code>name</code> is: <i>application name</i> Notice
<code>description</code>	Longer string to be displayed in the notice's body. The default is an empty string.

Property	Description
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to get the currently active notices that your application created. The `getNotices()` method is one of many methods that help you manage the notices you produce. You don't want to inundate your users with useless notices.

When you invoke the `addNotice()` method, an ID number is returned (that you can use when call `removeNotice()`). When a notice is dismissed, the `onNoticeEvent()` callback triggers with complete details. Finally, you can always find complete details regarding the existing notices any time by using the `getNotices()` method.

## Console.getPods()

### Availability

Macromedia Central.

### Usage

```
podData=shellReference.getPods()
```

### Example

```
// displays a list of all available pods and adds an option to remove
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    refreshListOfAllPods();

    myButton.onPress=function(){
        gShell.removePod(myListComponent.getSelectedItem().data);
        refreshListOfAllPods();
    };
};
```

```
refreshListofAllPods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of all the pods
    var allPods=gShell.getPods();

    // loop through the pods extracting their names and IDs
    for (var i=0; i<allPods.length; i++)
    {
        var thisPod=allPods[i];
        // add this pod's name and id to an MListBox component instance
        myListComponent.addItem(thisPod.name, thisPod.id);
    }
};
```

For additional examples of this method, see [AgentManager.removePod\(\)](#).

### Parameters

None.

### Returns

An array of `podData` structures for all pods available to this application. The values for a `podData` structure are set in the `product.xml` file or by a script calling `addPod()` or, in the case of `appId` and `id`, by Central itself. For reference, the entire `podData` object documentation is shown below.

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A `podData` object includes the following properties:

Property	Description
<code>id</code>	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
<code>name</code>	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.



Property	Description
<code>className</code>	<p>A string that specifies the name of the class, as defined in the product.xml file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your <code>pod</code>. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the product.xml file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<code>height</code>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<code>src</code>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the product.xml file's <code>pod</code> tag or <code>podClass</code> tag.)
<code>enabled</code>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<code>appid</code>	A number set by Central to associate a pod with your application.
<code>supportedTypes</code>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the product.xml file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<code>initialData</code>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

AgentManager, Console, or Shell method; gets the list of all pods available to this application. This includes pods listed in the product.xml file as well as any created by the `addPod()` method. To get a list of only those pods currently arranged in the Console, use the `getViewedPods()` method instead.

Realize that your application will only be listed once in the Console's pod selection pop-up menu (provided your application has at least one pod). For an application with more than one pod, the user will see a secondary pop-up menu inside the pod (adjacent to the pod's name). The `getPods()` method returns an array of all the pods that will appear in that secondary pop-up menu.

For more information see `getViewedPods()`, `addPod()`, and `viewPod()`.

## Console.getPreferences()

### Availability

Macromedia Central.

### Usage

```
prefObject=ref.getPreferences()
```

### Example

```
// Displays as customized a message as possible at startup
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // get all the preferences
    var all=gShell.getPreferences();

    // prepare a field to populate
    message_txt.text="";

    // encourage them to enable background tasks, just in case
    if(all.agentsEnabled==false)
    {
        message_txt.text+="Please enable background tasks."+newline;
    }

    // if we can't find first or last name, use a generic message
    if(all.userData.firstName==null || all.userData.lastName==null)
    {
        message_txt.text+="Welcome!"+newline;
    }
    else
    {
        message_txt.text+="Welcome "+all.userData.firstName+" "+
            all.userData.lastName+"."+newline;
    }

    // if the locations value isn't null
    if(all.locations!=null)
    {
        // store the location profile from the appropriate index
        var here=all.locations[all.currentLocationIndex];

        // fashion a personalized message to display
        message_txt.text+="You're probably glad to be "+
            here.label+" in beautiful "+here.city+".";
    }
};
```

## Returns

`pref0Object` Object containing details from the user's global preference settings. Depending on how much access the user has given to your application, you can find the values for some or all of the following properties.

Element	Description
<i>userData</i>	A structure with three properties: {firstName: xxx, lastName: xxx, email: xxx}
<i>locations</i>	An array of structures, each with the following properties: {label: xxx, address1: xxx, address2: xxx, city: xxx, state: xxx, zipcode: xxx, phone: xxx, country: xxx, latitude: xxx, longitude: xxx,}
<i>currentLocationIndex</i>	An index indicating the currently selected location (within the <code>locations</code> array).
<i>agentsEnabled</i>	A Boolean value that indicates whether agents are enabled.

## Description

AgentManager, Console, or Shell method; called by the pod or application to get the general Central preferences the user has exposed to your application. The value for the `agentsEnabled` property matches the user's setting for whether background tasks are allowed (set in the Advanced Preferences dialog box). This value is always available. In fact, you'll also see values for the `userData`, `locations`, and `currentLocationIndex` properties (based on settings under the Identity & Location Preferences dialog box). However, the values are all `null` by default and won't be available until the user has specifically allowed your application access to this data. It's easiest to visualize these properties and subproperties while viewing the Identity & Location Preferences dialog box.

## Console.getViewedApplications()

### Availability

Macromedia Central.

### Usage

```
arrayOfApplicationRecs=shellReference.getViewedApplications()
```

### Example

```
// this example stops users from launching multiple instances of your app
function onActivate(shell)
{
```

```

gShell=shell;

var activeApps = gShell.getViewedApplications();

if(activeApps.length>1)
{
    myAlertDialog("You're already running this app in another window");
}
};

```

### Parameters

None.

### Returns

An array of `applicationRecs` structures that contain details about each instance of your application. Each `applicationRec` structure has the following two properties:

Property	Description
<code>appId</code>	Number indicating the unique ID for your application. (All instances of your application share this number.)
<code>shellID</code>	Number indicating the unique ID for the particular shell running the application. (You can think of this as an ID for the Central shell.)

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of all the shell instances running your application. Naturally, this method only returns information about your applications. You can use the `getViewedApplications()` method to prevent users from launching multiple instances of your application (as the example shows). Also, if you develop a multi-window application, you can use the IDs gathered to set up unique `LocalConnection` channels (although it's often simpler to use the [LCService object](#)).

## Console.getViewedPods()

### Availability

Macromedia Central.

### Usage

```
arrayOfPodStructures=shellReference.getViewedPods()
```

### Example

```

// displays a detailed list of currently viewed pods
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our refresh function
    refreshListOfActivePods();
}

```

```

// give a button the ability to trigger our refresh function
myButton.onPress=function()
{
    refreshListOfActivePods();
};
};
refreshListOfActivePods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of the currently viewed pods

    var activePods=gShell.getViewedPods();

    // loop through the pods extracting some data for each one
    for (var i=0; i<activePods.length; i++)
    {
        var thisPod=activePods[i];
        var thisLabel=thisPod.podData.name+
            " (slot: "+thisPod.position+") "+
            ((thisPod.collapsed?"is not open":"is open"));

        myListComponent.addItem(thisLabel);
    }
};

```

## Parameters

None.

## Returns

An array of structures for each pod currently visible. The structures have the following properties:

Property	Description
viewerID	Number indicating a unique ID for the pod instance. This is the same number received as the third parameter in the pod's <code>onActivate()</code> handler.
position	Number indicating the current ordinal position of the pod (not the pixel location). Counting from the top, the uppermost pod is in <code>position 0</code> , then <code>position 1</code> , and so on.
collapsed	A Boolean value that indicates whether the pod is in the collapsed state.
podData	A Pod data object as specified in the <code>product.xml</code> file or defined when you call <code>addPod()</code> . For details on the properties contained in a <code>podData</code> object, see <code>getPods()</code> .

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of this application's currently *viewed* pods in the console. These are simply pods positioned in the Console (regardless of whether the console happens to be open). To get a list of all initialized pods initialized (that is, pods available to the user), regardless of whether they've been loaded in the Console, see [AgentManager.getPods\(\)](#) on page 154.

To determine whether the console is open, use [AgentManager.isConsoleOpen\(\)](#).

The `getViewedPods()` method only returns pods for your application.

While you can use the `removePod()` method (given the `id` property inside the `podData` property) this won't just close the pod but will actually remove it from the pods available to the user. There is no "close pod" method.

## Console.inLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.inLocalInternetCache(url);
```

### Example

```
// this example function loads images with the ultimate user control
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our homemade function
    loadImage("background.jpg");
}

loadImage=function(theImage, override)
{
    var imagePath="http://www.mysite.com/images/"+theImage;

    // if the image is already present (or they're overriding)
    if(gShell.inLocalInternetCache(imagePath)==true || override==true)
    {
        //add the image and load it into a clip instance
        error_txt.text="Loading "+theImage;
        gShell.addToLocalInternetCache(imagePath);
        clipInstance.loadMovie(imagePath);
    }
    else
    {
        // if they're connected
        if(gShell.isConnected()==true)
```

```

    {
        // get their approval by making the button set override to true
        error_txt.text="Do you want to download "+theImage+"?";
        okay_btn.onPress=function(){ loadImage(theImage,true) };
    }
    else
    {
        // if they're not connected just let them try again
        error_txt.text="Connect then press the okay button";
        okay_btn.onPress=function(){ loadImage(theImage)};
    }
}
};

```

### Parameters

*url* A string; fully qualified path that provides the location of the file to be added to the local Internet cache.

### Returns

A Boolean value; `true` if the URL is in the local Internet cache, `false` if not found.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to check if a URL is in the local Internet cache. There are several strategies that might require this method. Before requiring a user to endure a long download, you can first check if the file is available locally, in which case, you get the user's approval first. Also, if the user has indicated that they're not online and the file is not available locally, you can tell them they need to go online first. Finally, the `inLocalInternetCache()` method provides an indirect way to confirm that attempts to add files to the local Internet cache are successful, as described in the following best practice.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

It's a good idea to always confirm the success of any `addToLocalInternetCache()` call by immediately issuing the `inLocalInternetCache()` method with an appropriate follow-up action if the method returns a value of `false`. This is because if the user's 20 MB cache is exceeded and they don't allow an increase, then `addToLocalInternetCache()` effectively fails.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

```

http://www.mydomain.com/pub/myFile.swf
http://applications.mydomain.com/pub/myFile.swf

```

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

## Console.isConnected()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.isConnected()
```

### Example

```
// this example checks the connection state at startup
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // take current state and trigger onNetworkChange (where our code resides)
    this.onNetworkChange(gShell.isConnected());
};

onNetworkChange=function(connected)
{
    // save the connection state in a variable
    gOnline=connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;
};
```

### Parameters

None.

### Returns

A Boolean value: `true` if the user is connected, `false` if offline.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to determine current network status. You should consolidate all your code related to connectivity inside the `onNetworkChange()` handler. Although Central triggers `onNetworkChange()` automatically, it only does so when the connection status *changes*. Therefore, you need to use `isConnected()` initially to bring your application in sync. The example shows how an application can trigger its own `onNetworkChange()` handler (though usually Central does this). This way, all the code is consolidated in one place. There's no reason to repeatedly check `isConnected()` from multiple places in your code.



## Console.loadApplication()

### Availability

Macromedia Central.

### Usage

```
shellReference.loadApplication([ initialData ]);
```

### Example

```
//this example provides a button in the pod to launch the main app
onActivate=function(shell)
{
    //set a variable to reference the Console
    gShell=shell;

    //set up a button to open our application
    myToss_btn.onClick = function()
    {
        gShell.loadApplication("from_pod");
    };
};
```

### Parameters

*initialData* Any data type; optional parameter providing initial data to be passed into the application when loading. (Received in the application's `onSelectedItem()` callback.)

### Returns

Nothing.

### Description

Console event handler; your pod can call it to load its parent application. Additionally, you can pass data using the optional parameter. You can use this parameter to pass data to your application. If the application is not open, Central opens it, subject to normal application opening rules. If the application is already open, it does not open a second window, but gets an `onSelectedItem()` call.

The optional parameter is also useful when attempting to send data from a pod using the Blast feature. However, because the Console doesn't have a Blast menu, you may want to send any data that is to be sent using the Blast feature by way of `loadApplication()`, and have code in the application's `onActivate()` callback proceed to fashion the `selectedItem` data.

Macromedia recommends that you launch applications only when the user deliberately selects an option to do so. This is similar to the recommendation against automatically viewing pods. Additionally, when the `loadApplication()` will be triggering a Blast operation, you should use the standard "Send" curved arrow icon (available in the Central Components). The important thing is to give the user control.

For more information on the Blast feature, see [Chapter 7, "Using the Blast Feature,"](#) on page 105.

## Console.newLocationDialog()

### Availability

Macromedia Central.

### Usage

```
shellReference.newLocationDialog([reqFields])
```

### Example

```
// this prompts the user to create a new location with city and state required
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    newLocation_btn.onPress=function()
    {
        gShell.newLocationDialog(["locCity", "locState"]);
    }
};
```

### Parameters

*reqFields* An array that contains strings for the fields that you want to designate as required. The following table lists valid field names. It's easiest to visualize these field names while viewing the Identity & Location Preferences dialog box. (You can pass the literal string "noDialog" to open the Identity & Location Preferences dialog box without creating a new location entry or designating any required fields.)

Field	Description
"firstName"	User's first name
"lastName"	User's last name
"email"	User's e-mail address
"locAddress1"	The first line of the user's address
"locAddress2"	The second line of the user's address
"locCity"	User's city
"locState"	User's state
"locZip"	User's zip code
"locPhone"	User's phone number
"locLat"	User's latitude
"locLong"	User's longitude

### Returns

Nothing.

## Description

Shell or Console method; called by your application or pod to open the New Location dialog box from the user's preferences, so that they may create a new location and then edit the values. This is the same as if the user selects New Location from the Location pop-up menu in the Identity & Location section of the general preferences. The only difference here is that the user never sees the rest of their preferences if they cancel the operation.

To make no fields required, don't pass anything. To make the Identity & Location Preferences dialog box appear without creating a new location entry or designating any required fields, pass the literal string "noDialog".

Additionally, the `newLocationDialog()` method lets you designate that any or all fields are required. The user sees a small page curl on the required fields, and red lines around any fields they attempt to leave blank. (Be sure to pass an array containing *strings* that match the values listed in the table.)

## Console.removeFromLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
shellReference.removeFromLocalInternetCache(URL)
```

### Example

```
// this example attempts to free up space in the user's cache when appropriate
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // add a new image to the local cache and load it into a clip
    var imagePath="http://www.mysite.com/images/";
    var theImage="photo.jpg";
    gShell.addToLocalInternetCache(imagePath+theImage);
    clipInstance.loadMovie(imagePath+theImage);

    // if the new image isn't present that means the cache is full
    if(gShell.inLocalInternetCache(imagePath+theImage)==false)
    {
        // take a list of previously loaded images (could be dynamic)
        myImageList=["big1.jpg", "big2.jpg", "big3.jpg"];

        // and remove each one
        for(var i=0;i<myImageList.length;i++)
        {
            gShell.removeFromLocalInternetCache(imagePath+myImageList[i]);
        }
    }
};
```

## Parameters

*url* Fully qualified location of the file to be removed from the local Internet cache.

## Returns

Nothing.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a URL from the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the web rather than from the cache.

The file in question must be *in* the local Internet cache for this method to work. That means that previously an application must have issued `addToLocalInternetCache()`. Another way to remove a file from the local Internet cache is by setting an expiration date when invoking `addToLocalInternetCache()`. Finally, the `Shell.addToLocalInternetCache()` method also has an `overwrite` parameter which effectively removes a file by replacing it. For more information, see `addToLocalInternetCache()`.

## Console.removeNotice()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.removeNotice(noticeID)
```

### Example

```
// this example creates 3 notices and removes all 3 when any one is dismissed
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // create an array to store IDs for the notices we create
    gPostedNotes=new Array();
    //create a notice, set its name, and add it
    var noticeData = new Object();
    noticeData.engageString="Remove";
    noticeData.name = "One";
    gPostedNotes.push( gShell.addNotice(noticeData) );
    noticeData.name = "Two";
    gPostedNotes.push( gShell.addNotice(noticeData) )
    noticeData.name = "Three";
    gPostedNotes.push( gShell.addNotice(noticeData) )
};

onNoticeEvent=function(event, noticeData, initialData)
{
    // while gPostedNotes still has items remaining
```

```

while(gPostedNotes.length>0)
{
    // pop one off and remove it
    var thisNotice=gPostedNotes.pop();
    gShell.removeNotice(thisNotice);
}
};

```

### Parameters

*noticeID* A number identifying the specific notice you want removed. You can use the number returned when you create a notice using the `addNotice()` method. You can also use an `id` property of any object within the array of notices objects returned from the `getNotices()` method.

### Returns

A Boolean value: `true` if the notice was removed, otherwise `false`.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a notice. You can only remove notices that your application created.

The most direct way to track IDs is to store them as you use `addNotices()`. However, this can be difficult because users can dismiss notices (in which case you'll have to trap the `onNoticeEvent()` event) and they might leave the notices untouched when they quit Central (in which case you'll have to save a LocalShared object). Probably the easiest tracking method is to include initial data in the second parameter of your `addNotice()` call. Use `getNotices()` and then step through each item returned looking for a matching `appData` property.

A best practice is to minimize the total number of notices by first deleting old notices and then replacing them with new ones containing up-to-date information. For an example of this practice, see [Shell.getNotices\(\)](#).

## Console.removePod()

### Availability

Macromedia Central.

### Usage

```
shellReference.removePod(id)
```

### Example

```

// product.xml excerpt from within the <application> tag:
<podclass name="myRegularPod" src="pod.swf"/>
<podclass name="mySpecialPod" src="specialpod.swf"/>

<pod name="myDefaultPod" className="myRegularPod" />

// this example temporarily exposes a special pod for the user to open
onActivate=function(shell)

```

```

{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // give the user the opportunity to access the special pod class
    gCounter=0;
    my_btn.onPress=function()
    {
        gCounter++;
        if(gCounter==3)
        {
            // add a new pod
            var podData=new Object();
            podData.name="special #3";
            podData.className="mySpecialPod";
            gSpecialPodID=gShell.addPod(podData);
        }
        if(gCounter==4)
        {
            // remove the special pod
            gShell.removePod(gSpecialPodID);
        }
    }
};

```

### Parameters

*id* A number returned when calling the `addPod()` method to create a pod. You can also use an `id` property of any object within the array of objects returned to the `getPods()` method.

### Returns

Nothing.

### Description

`AgentManager`, `Console`, or `Shell` object method; called by an agent, pod, or application, respectively, to remove a pod. Although this method removes from view any pods currently arranged in the Console, it works differently from the way the user manually closes a pod. In the case of `removePod()`, you aren't removing an individual pod SWF instance, but rather removing a pod instance once associated with your application. Naturally, if a matching pod is present in the Console, it must be closed, but using `removePod()` means the user will no longer be able to add that pod instance manually. There is no "close pod" method.

For more information about pods and pod classes see the entry for [Console.addPod\(\)](#). For more information about the difference between application pods and currently viewed pods, see the entries for [Console.getPods\(\)](#) and [Console.getViewedPods\(\)](#), respectively.

Central automatically removes all pods associated with an application when a user uninstalls the application.

## Console.startAgent()

### Availability

Macromedia Central.

### Usage

```
shellReference.startAgent()
```

### Example

```
// this example stops and starts the agent as connection status changes
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;
};

onNetworkChange=function(connected)
{
    if(connected==false)
    {
        gShell.stopAgent();
    }
    else
    {
        gShell.startAgent();
    }
};
```

### Parameters

None.

### Returns

A Boolean value: `true` if the agent was started, `false` if offline.

### Description

Console or Shell method; called by a pod or an application to start the agent associated with your application as defined in the `product.xml` file. The `product.xml` file also lets you set your agent to start automatically every time Central starts. Simply set the `started` attribute to `true` inside the agent tag, as the following code shows:

```
<agent name="myAgent" src="agent.swf" started="true"/>
```

Most often, you use the `startAgent()` method after you issue a `stopAgent()` call.

## Console.stopAgent()

### Availability

Macromedia Central.

## Usage

```
myBoolean=shellReference.stopAgent()
```

## Example

```
// this example stops and starts the agent as connection status changes
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
};

onNetworkChange=function(connected)
{
    if(connected==false)
    {
        gShell.stopAgent();
    }
    else
    {
        // this only works from an application or pod (not agent)
        gShell.startAgent();
    }
};
```

## Parameters

None.

## Returns

A Boolean value: true if the agent was successfully stopped, otherwise false.

## Description

AgentManager, Console, or Shell method; called by an agent itself, a pod, or an application to stop the agent SWF file (listed in the product.xml file). After it stops, the agent won't start again until you call `startAgent()` from an application or pod instance. In fact, agents do not start automatically unless the product.xml file's agent tag includes the attribute `started="true"`.

**Note:** Your application can only have one agent.

A best practice is to keep as much code as possible in your agent. In such a case, the `stopAgent()` method has questionable value. However, if your agent is primarily executing background tasks using `setInterval()`, it might be easiest to first clear all the intervals and then simply call `stopAgent()`. Additionally, if you design an application to use the agent only temporarily, then it makes sense to use `stopAgent()` (and `startAgent()`).

**Note:** You don't need to remove agents when the user uninstalls your application; Central does this for you.



## Console.viewPod()

### Availability

Macromedia Central.

### Usage

```
shellReference.viewPod( podID [, bForce] )
```

### Example

```
// This example adds a pod only after the user first runs the app.  
// It requires you to define a PodClass named "post_install" in the product.xml
```

```
onActivate=function(shell)  
{  
    // set a variable to reference the AgentManager, Shell, or Console  
    gShell=shell;  
  
    // check to see if the pod is installed  
    var foundID=null;  
    var allPods=gShell.getPods();  
    for(var i=0;i<allPods.length;i++)  
    {  
        if(allPods[i].className=="post_install")  
        {  
            foundID=allPods[i].id;  
            gPodID=foundID;  
            break;  
        }  
    }  
  
    // create it if it wasn't found  
    if(foundID==null)  
    {  
        var podData=new Object();  
        podData.name="Post Install Pod";  
        podData.className="post_install";  
        gPodID=gShell.addPod( podData);  
    }  
  
    // give the user a way to view the pod  
    podOpen_btn.onPress=function()  
    {  
        // should first use getViewedPods to avoid multiple views of this pod  
        //as is, this will view the pod added or found above  
        gShell.viewPod(gPodID);  
    };  
};
```

### Parameters

*podID* The number returned when calling `addPod()` to create a pod. You can also use an `id` property of any object within the array of objects returned to the `getPods()` method.

*bForce* Optional parameter that forces the creation of a new pod viewer in the Console. The *bForce* parameter is a Boolean value. If *true* (or omitted), `viewPod()` forces the creation of a new pod viewer in the Console. If *false*, a new pod viewer is created only if the pod referred to by `podID` is not already viewed in the Console (so no duplicate pods appear).

### Returns

Nothing.

### Description

AgentManager, Console, or Shell method; called by an agent, application, or pod, respectively, to open a specific pod instance so that it becomes visible in the top slot of the Console. All of the other visible pods are moved down in the console.

To call `viewPod()` you need a *podID* parameter. This means that you either must have used a script to call `addPod()`, which returns an *id*, or used the `getPods()` method to get an array full of structures (each one including an *id*). Using the `getPods()` method might seem to be haphazard if your application has more than one pod, as you wouldn't know which item in the array was for which pod. However, the data in the array returned from `getPods()` includes other details, including any initial data that you can specify in the `product.xml` file. It is probably most intuitive to simply use `viewPod()` immediately after the creation of a pod by using the `addPod()` method, as the example shows.

A best practice is to only call the `viewPod()` method in response to a direct user action. The user should choose how to best populate the Console.

## DataProviderClass object

**ActionScript Class Name** `mx.central.data.DataProviderClass`

The `DataProviderClass` object provides a robust and generalized interface for creating and managing a wide variety and number of data items. This object contains a detailed event model that broadcasts granular `modelChanged` events to any listening objects.

All list-based components (`DataGrid`, `ComboBox`, `ListBox`, and others) implement one or more `DataProviderClass` objects to manage their ordered item lists.

A `DataProviderClass` object can and should be used to work efficiently with ordered data sets, even if the set will never be presented to a component.

You should become familiar with the [LCDDataProvider object](#), because the `LCDDataProvider` object is nearly identical to the `DataProviderClass` object. `LCDDataProvider` provides access to the synchronous Local Connection Object.

## Extending the DataProviderClass object

You can create a `DataProviderClass`-compliant object by implementing all the methods and properties described in this document. A List-based component, such as the `ListBox` component, could then use that object as a data provider. To extend the `DataProviderClass` object, be sure to call its `init()` function when the subobject is instantiated.

The following example shows how you can extend the `DataProviderClass` object:

```
SomeNewClassName = function() {
    this.init();
}
SomeNewClassName.prototype = new DataProviderClass();

SomeNewobjectsName.prototype.init = function() {
    super.init();
}

SomeNewClass.prototype.updateView = function(view, eventObj) {
    if(view.modelChanged != undefined) {
        view.modelChanged(eventObj);
    } else {
        view.onSomeEventTrigger(eventObj);
    }
}

this.onSomeEventTrigger = function(eventObj) {
    trace(">> ON SOME EVENT TRIGGER CALLED");
    for(var i in eventObj) {
        trace(i + " : " + eventObj[i] + " type : " + typeof(eventObj[i]));
    }
}

var myNewClass = new SomeNewClass();

myNewClass.addItem({label:"someLabel1", data:"someData1",
    sortable:"SOMEDATA1"});
myNewClass.addItem({label:"someLabel2", data:"someData2",
    sortable:"SOMEDATA2"});
myNewClass.addItem({label:"someLabel3", data:"someData3",
    sortable:"SOMEDATA3"});
myNewClass.sortItemsBy("sortable", "DESC");

myNewClass.addListener(this);
this.someListBox.dataProvider = myNewClass;
```

## Method summary for the `DataProviderClass` object

The following table summarizes the methods for the `DataProviderClass` object:

Method	Description
<code>DataProviderClass.addItem()</code>	Adds a single item, <i>item</i> , to the end of the list of items.
<code>DataProviderClass.addItemAt()</code>	Adds a single item at a specific index in the list of items.
<code>DataProviderClass.addItems()</code>	Adds a set of item objects to the end of the list of items. This set can be either an array of objects (ordered) or an object of objects (unordered).

Method	Description
<code>DataProviderClass.addItemAt()</code>	Adds a set of item objects to a specific index in the list of items.
<code>DataProviderClass.addListener()</code>	Passes a reference to an object that is added to the <code>DataProviderClass._listeners</code> array, and determines whether to call the object's <code>modelChanged</code> event.
<code>DataProviderClass.getAllItems()</code>	Returns the entire items array. This method returns only a reference to the actual <code>dataProviderInstance.items</code> array.
<code>DataProviderClass.getIndexByKey()</code>	Returns the index of the first item (starting from the beginning of the list) whose property indicated by <i>key</i> matches the value passed in.
<code>DataProviderClass.getIndicesByKey()</code>	Returns an array of indexes for items whose property specified by <i>key</i> matches <i>value</i> .
<code>DataProviderClass.getItemAt()</code>	Returns the item at <i>index</i> .
<code>DataProviderClass.getItemByKey()</code>	Returns the first item object found whose property specified in <i>key</i> matches <i>value</i> .
<code>DataProviderClass.getItemID()</code>	Returns the value of the <code>item._ID_</code> property found at <i>index</i> .
<code>DataProviderClass.getItemsByKey()</code>	Returns an array of objects whose property specified in <i>key</i> matches <i>value</i> .
<code>DataProviderClass.getLength()</code>	Returns the total number of items in the data provider.
<code>DataProviderClass.getSortState()</code>	Returns a sort object with two string properties: <code>sortField</code> and <code>order</code> .
<code>DataProviderClass listener.modelChanged()</code>	Broadcasts when the data provider changes
<code>DataProviderClass.removeAll()</code>	Removes all items in the data provider.
<code>DataProviderClass.removeItemAt()</code>	Removes the item at <i>index</i> and returns it.
<code>DataProviderClass.removeListener()</code>	Searches the <code>_listeners</code> array for the reference passed in, and removes it if found.
<code>DataProviderClass.replaceAllItems()</code>	Deletes and replaces all of the items in the <code>DataProviderClass</code> instance.
<code>DataProviderClass.replaceItemAt()</code>	Overwrites an item at the specified index with a new item object.
<code>DataProviderClass.setItemByKey()</code>	Overwrites an existing item object using the key of any existing item property.
<code>DataProviderClass.sort()</code>	Sorts the items using a custom function, similar to <code>Array.sort</code> , but lets you pass in additional arguments to persist and retrieve user-driven sort selections.

Method	Description
<code>DataProviderClass.sortItemsBy()</code>	Sorts the items in the specified order, using the built-in <code>Array.sortOn</code> method with the item property specified in <code>key</code> .
<code>DataProviderClass.updateItem()</code>	Overwrites an existing item object with a new item object.
<code>DataProviderClass.updateItemByIndex()</code>	Overwrites an existing item object at <i>index</i> with a new item object.
<code>DataProviderClass.updateView()</code>	Broadcasts events, other than the <code>modelChanged</code> event, for any objects that extend the <code>DataProviderClass</code> object. This method is called for each <code>_listener</code> that is added when the <code>DataProviderClass</code> instance triggers an event.

## Events for the DataProviderClass object

By extending the `DataProviderClass` object and overriding the `updateView` method, you can make the data provider broadcast to any method you define. All components listen for `modelChanged` events, so you must broadcast to the `modelChanged` event, or your implementation will not work with existing components. To broadcast to any method, pass in an `eventObject` as described next for the `modelChanged` event:

Event	Description
<code>DataProviderClass listener.modelChanged()</code>	Broadcasts when the data provider changes.

## Constructor for the DataProviderClass object

If your application is working with any type of data, the data can usually be stored in a `DataProviderClass` object. Data stored in a `DataProviderClass` object is much easier to access, transmit, and manipulate.

However, if your application has an extremely large data set, the performance of the `DataProviderClass` object might not be optimal, especially if you use custom sort functions. If the order of the data set is not important, you might want to store the data in an unordered, keyed list instead.

The following example shows how to instantiate a data provider for a list box:

```
// For a list box
var myDp = new mx.central.data.DataProviderClass();
myDp.addItem({label:"someLabel1", data:"someData1"});
myDp.addItem({label:"someLabel2", data:"someData2"});
myDp.addItem({label:"someLabel3", data:"someData3"});
this.someListBox_mc.dataProvider = myDp;
```

## DataProviderClass.addItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItem(item)
```

### Parameters

*item* An object; the item to add.

### Returns

Nothing.

### Description

Method; adds a single item, *item*, to the end of the list of items. If the item has a property named `_ID_`, that property is overwritten with a new, unique value.

### Example

The following example populates a data provider and adds an item, with a label property, to the end of the data provider:

```
this.dP = new mx.central.data.DataProviderClass();

function populateDp () {
    setData();
    grid.dataProvider = this.dP;
    statusField.text = "Click AddItemsAt button first time and AddItems the
    second time.";
}

function setData(){
    for (var i=0; i<6; i++) {
        this.dP.addItem( {label:"first "+i, data:"second"});
    }
}
```

## DataProviderClass.addItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItemAt(index, item)
```

## Parameters

*index* The index at which to add the item.

*item* The item to add.

## Returns

Nothing.

## Description

Method; adds a single item at a specific index in the list of items. If the item has a property named `_ID_`, that property is overwritten with a new, unique value.

## Example

The following example adds an item, with a label property, to the data provider `myDP` at the fourth position:

```
myDP.addItemAt(3, {label: "this is the fourth Item"});
```

## DataProviderClass.addItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItem(items)
```

## Parameters

*items* An array of objects (ordered) or an object of objects (unordered).

## Returns

Nothing.

## Description

Method; adds a set of item objects to the end of the list of items. This set can be either an array of objects (ordered) or an object of objects (unordered). Either way, the items are stored in a specific order once added. If any item has a property named `_ID_`, that property is overwritten with a new, unique value.

## Example

The following code adds 20 items to the end of the list of items:

```
this.dP = new mx.central.data.DataProviderClass();

function gridAddItems() {
    var object2 = new Object();
    for (var i=0; i<20; i++) {
```

```

        object2[i] = {label: "new first" + (++this.incr), data: "new second"};
    }
    dP.addItem(object2);
    statusField.text = "Verify that 20 items are added to the end of the
    dataprovider.";
}

```

## DataProviderClass.addItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItemAt(index, items)
```

### Parameters

*index*    The number at which to add the items.

*items*    An array of objects (ordered) or an object of objects (unordered).

### Returns

Nothing.

### Description

Method; adds a set of item objects to a specific index in the list of items. If any item has a property named `_ID_`, that property is overwritten with a new, unique value.

### Example

The following code provides an example of how to populate a data provider and use the `addItemAt()` method:

```

this.dP = new mx.central.data.DataProviderClass();

function populateDp () {
    setData();
    grid.dataProvider this.dP;
    statusField.text = "Click AddItemsAt button first time and AddItems the
    second time.";
}

function gridAddItemsAt() {
    var object1 = new Object();
    for (var i=0; i<100; i++) {
        object1[i] = {label: "new first" + (++this.incr), data: "new second"};
    }

    dP.addItemAt(2, object1);
}

```



```
        statusField.text = "100 new rows should have been added after row 2.\nClick  
clearDp button to clear the dataprovider.";  
    }
```

## DataProviderClass.addListener()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addListener(view [,doNotUpdate])
```

### Parameters

*view* A reference to an object.

*doNotUpdate* A Boolean value. If true, the item's `modelChanged` event is not triggered. If null, undefined, or false, a `modelChanged` event is triggered on the referenced object.

### Returns

Nothing.

### Description

Method; passes a reference to an object that is added to the `DataProviderClass._listeners` array, and determines whether to call the object's `modelChanged` event.

If *doNotUpdate* is true, the item's `modelChanged` event is not triggered. If *doNotUpdate* is null, undefined, or false, this method triggers a `modelChanged` event on the reference. An event object with an event of "updateAll" is passed to the `modelChanged` event.

## DataProviderClass.getAllItems()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getAllItems()
```

### Parameters

None.

### Returns

A reference to an array.

## Description

Method; returns the entire items array. This method returns only a reference to the actual *dataProviderInstance.items* array. Any changes made directly to the returned array will likely cause problems in the data provider. Use this accessor method for read-only purposes.

## Example

The following example traces the reference returned:

```
var aItems:Array = myDP.getAllItems();
```

## DataProviderClass.getIndexByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getIndexByKey(key, value)
```

### Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

### Returns

An integer, or, if no items are found, -1.

## Description

Method; returns the index of the first item (starting from the beginning of the list) whose property indicated by *key* matches the *value* passed in.

## Example

The following example gets the index of the item added:

```
// adds two items to the list
someDp.addItem({category:"recipes", name:"salads"});
someDp.addItem({category:"recipes", name:"desserts"});

// returns the index of the first item encountered with the matching key/value
pair
var myIndex = someDp.getIndexByKey("category", "recipes");
trace(myIndex);
```

## DataProviderClass.getIndicesByKey()

### Availability

Macromedia Central Player.

## Edition

Macromedia Central SDK.

## Usage

```
getIndicesByKey(key, value)
```

## Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

## Returns

An array of indexes or, if no items are found, an empty array.

## Description

Method; returns an array of indexes for items whose property specified by *key* matches *value*.

## Example

The following example gets the indexes of the items added:

```
// adds two items to the list
someDp.addItem({category:"movies", name:"thrillers"});
someDp.addItem({category:"movies", name:"comedies"});

// returns the indexes of the first item encountered with the matching key-
// value pair
var myIndexes = someDp.getIndexByKey("category", "movies");
trace(myIndexes);
```

## DataProviderClass.getItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemAt(index)
```

### Parameters

*index* The index of the item to get.

### Returns

An object, or, if an invalid index is used, undefined.

### Description

Method; gets the item at *index* and returns the item.

## Example

The following code adds two items to the list and gets the second item:

```
myDp.addItem("label0", "data0");
myDp.addItem("label1", "data1");
var myItem:Object = myDp.getItemAt(1);
```

## DataProviderClass.getItemByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemByKey(key, value)
```

### Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

### Returns

An object or, if no item is found, null.

### Description

Method; returns the first item object found whose property specified in *key* matches *value*.

## Example

The following example adds two items and returns the first item:

```
myDp.addItem({category:"circus", name:"tiger"});
myDp.addItem({category:"circus", name:"lion"});
// returns the first object that matches the property key
// (in this case, the item named "tiger")
var myItem:Object = myDp.getItemByKey("category", "circus");
```

## DataProviderClass.getItemID()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemID(index)
```

## Parameters

*index* The index of the item whose index is retrieved.

## Returns

A string.

## Description

Method; returns the value of the `item._ID_` property found at *index*. The returned value is a string data type but can be evaluated to a number.

## Example

The following example gets the ID of the third item in the list:

```
var itemID:String = getItemID(2);
```

## DataProviderClass.getItemsByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemsByKey(key, value)
```

### Parameters

*key* A string.

*value* A primitive data type value.

### Returns

An array of objects, or, if no items are found, an empty array.

### Description

Method; returns an array of objects whose property specified in *key* matches *value*. The items returned are in the same order (relative to one another) as in the data provider.

### Example

The following example adds two items to the list and returns them both:

```
// returns both objects that match the property key
myDp.addItem({category:"hobbit", name:"frodo"});
myDp.addItem({category:"hobbit", name:"bilbo"});
var matchingItems:Array = myDp.getItemsByKey("category", "hobbit");
```

## DataProviderClass.getLength()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getLength()
```

### Parameters

None.

### Returns

An integer.

### Description

Method; returns the total number of items in the data provider.

### Example

The following example traces the total number of items in the list:

```
var qty:Number = myDp.getLength();  
trace("There are " + qty + " items in the list.");
```

## DataProviderClass.getSortState()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getSortState()
```

### Parameters

None.

### Returns

A sort object with two string properties.

### Description

Method; returns a sort object with two string properties, *sortField* and *order*:

- *sortField* is the property name last used for sorting or last passed to the `sort` method.

- *order* is the order property. Its value depends on what, if anything, was last passed to the `sort` or `sortItemsBy` methods. Possible values are ascending, "ASC"; descending, "DESC"; or an empty string, "".

If the properties have no value, they still show as empty strings.

### Example

The following example gets and displays the sort state for a `DataProviderClass` instance:

```
var sortObject:Object = myDP.getSortState();
trace("sortField=" + sortObject.sortField);
trace("order=" + sortObject.order);
```

## DataProviderClass *listener.modelChanged()*

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
listenerObject = new Object();
listenerObject.modelChanged = function(eventObject) {
    // insert your code here
}
myDataProvider.addListener(listenerObject)
```

### Parameters

*eventObject*    An object with additional properties.

### Returns

Nothing.

### Description

Event; broadcast when the data provider changes. The `modelChanged()` method is not a method of the `DataProviderClass`; it is a function called *by* the `DataProviderClass` on an object registered as a view or listener object.

The *eventObject* object has the following properties:

- *source*    A reference back to the data provider instance that initiated the event.
- *event*    A string identifying the type of change made so that any listening views can perform granular updates. The following strings are possible values: "updateAll", "updateRows", "deleteRows", "sort", or "addRows".
- *firstRow*    The number of the first row affected by the change.
- *lastRow*    The number of the last row affected by the change.

**Note:** The *firstRow* and *lastRow* parameters are the same if only one row was affected. They are omitted if the event is "sort" or "updateAll".

## DataProviderClass.removeAll()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeAll()
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; removes all items in the data provider.

### Example

The following code removes all items from a DataProviderClass instance:

```
myDp.removeAll();
```

## DataProviderClass.removeItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeItemAt(index)
```

### Parameters

*index* The index of the item to remove.

### Returns

An object.

### Description

Method; removes the item at *index* and returns the removed item.

### Example

The following code removes the third item in the list:

```
var removedItem:Object = myDp.removeItemAt(2);
```



## DataProviderClass.removeListener()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeListener(listenerObj)
```

### Parameters

*listenerObj* A reference to an object.

### Returns

Nothing.

### Description

Method; searches the `_listeners` array for the reference passed in, and removes it if found.

### Example

The following code removes a listener from the `DataProviderClass` instance named `myDp`:

```
myDp.removeListener(myListener);
```

## DataProviderClass.replaceAllItems()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
replaceAllItems(items)
```

### Parameters

*items* An array of objects (ordered) or an object of objects (unordered).

### Returns

Nothing.

### Description

Method; deletes all of the items in the list and replaces them with the items specified in the *items* parameter.

## DataProviderClass.replaceItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
replaceItemAt(index, item)
```

### Parameters

*index* The index of the item to be replaced.

*item* An object.

### Returns

Nothing.

### Description

Method; overwrites an item at the specified index with a new item object. If the item passed in has an `_ID_` property, that property is overwritten with a new one.

### Example

The following code replaces the fourth item in the list with the new item:

```
myDp.replaceItemAt(3, {category: "circus" , name: "lion"});
```

## DataProviderClass.setItemByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
setItemByKey(key, item)
```

### Parameters

*key* A string.

*item* An object.

### Returns

The index of the overwritten item.

## Description

Method; overwrites an existing item object using the key of any existing item property. The *key* specified is used to locate the existing item by matching that property on the item specified in *item*.

## Example

The following code adds an item at the fourth position then overwrites the item using the property key:

```
myDp.addItemAt(3, {label : "bear"});  
myDp.setItemByKey("bear", {label: "tiger"});
```

## DataProviderClass.sort()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
sort(sortFunction[,key, order])
```

### Parameters

*sortFunction* A function.

*key* A string that should map to an item property that you have defined.

*order* A string, "ASC" for ascending order or "DESC" for descending order.

### Returns

Nothing.

## Description

Method; sorts the items in a list using a custom function, similar to `Array.sort`, but lets you also pass in additional parameters to retrieve sort selections made by the user and make them persistent.

**Note:** This method uses ActionScript to execute the sort, which is inherently slow. For large data sets, you might want to use the `sortItemsBy` method, which uses the native C++ implementation of the `Array.sortOn` method.

## Example

The following code sorts the list by the `name` property in ascending order, using a predefined sort function named `mySortFunction`:

```
myDp.sort(mySortFunction, "name", "ASC");
```

## DataProviderClass.sortItemsBy()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
sortItemsBy(key, order)
```

### Parameters

*key* A string that should map to an item property that you have defined.

*order* A string, "ASC" for ascending order or "DESC" for descending order.

### Returns

Nothing.

### Description

Method; sorts the items in the specified order, using the built-in Array.sortOn method, with the item property specified in *key*.

This method works only for items that have a primitive data type in the specified key.

**Note:** This method uses the native C++ implementation of the Array.sortOn method. That implementation sorts items by their ASCII codes, which means that all lowercase characters appear before uppercase characters. For example, lowercase “z” is placed before uppercase “A”. To sort in proper alphabetical order, you can create a new item property that has each item in uppercase letters, and sort on that property. This implementation will likely be faster than using a custom sort function.

### Example

The following example sorts items by their `name` property in ascending order:

```
myDp.sortItemsBy("name", "ASC");
```

## DataProviderClass.updateItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateItem(item)
```

### Parameters

*item* The existing object that overwrites the internal item.

### Returns

The index into which the item was passed.

### Description

Method; passes an existing item object to the data provider to overwrite an internal item with the same `_ID_` property.

## **DataProviderClass.updateItemByIndex()**

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateItemByIndex(index, item)
```

### Parameters

*index* The index of the item to be overwritten.

*item* The new item object.

### Returns

The index of the overwritten item.

### Description

Method; overwrites the existing item object at *index* with a new item object. If the item object passed in does not have an `_ID_` property, an `_ID_` property is added to the new item object.

## **DataProviderClass.updateView()**

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateView(view, eventObject)
```

### Parameters

*view* A reference to an object.

*eventObject* A valid event object.

### Returns

Nothing.

## Description

Method; broadcasts events. Use to broadcast events other than the `modelChanged` event for any objects that extend the `DataProviderClass` object. This method is called for each `_listener` that is added when the data provider triggers an event. The method currently contains one line of code: `view.modelChanged(eventObj);`

## FileReference object

**ActionScript Class Name** `FileReference`

Central applications can access local files using the `FileReference` object. `FileReference` objects refer to a single file on the user's local disk.

## Method summary for the FileReference object

Method	Description
<code>FileReference.browse()</code>	Initializes the <code>FileReference</code> object by prompting the user to select a file anywhere on the local disk.
<code>FileReference.close()</code>	Closes the file and finishes all read and write operations.
<code>FileReference.copy()</code>	Creates a copy of the file.
<code>FileReference.copyIntoCache()</code>	Creates a copy of the file in the local Internet cache.
<code>FileReference.create()</code>	Initializes the <code>FileReference</code> object by creating a file in the user's local Internet cache.
<code>FileReference.deleteFile()</code>	Deletes the file.
<code>FileReference.download()</code>	Initializes the <code>FileReference</code> object by downloading a file from the Internet and saving it on the user's local disk.
<code>FileReference.exists()</code>	Checks to see whether a file exists in the user's local Internet cache.
<code>FileReference.getPosition()</code>	Gets the current read or write position in the file.
<code>FileReference.locate()</code>	Displays the file using Windows Explorer or Macintosh Finder.
<code>FileReference.move()</code>	Moves a file to a new location.
<code>FileReference.open()</code>	Initializes the <code>FileReference</code> object by opening a file in the user's local Internet cache.
<code>FileReference.readBytes()</code>	Reads a specified number of bytes from the file into an array.
<code>FileReference.readFile()</code>	Reads the entire contents of a file into a string.
<code>FileReference.readString()</code>	Reads a specified number of bytes from the file into a string.
<code>FileReference.rename()</code>	Renames the file.
<code>FileReference.saveAs()</code>	Saves the file to a new location outside of the local Internet cache.
<code>FileReference.setPosition()</code>	Sets the current read or write position in the file.
<code>FileReference.upload()</code>	Uploads the file using HTTP POST.

Method	Description
<code>FileReference.writeBytes()</code>	Writes an array of bytes to the file.
<code>FileReference.writeString()</code>	Writes a string to the file.

## Property summary for the FileReference object

Property	Description
<code>FileReference.name</code>	The name of the file.
<code>FileReference.creationDate</code>	The date the file was created.
<code>FileReference.creator</code>	The Macintosh 4 character creator string; undefined on Windows.
<code>FileReference.modificationDate</code>	The date the file was last modified.
<code>FileReference.readOnly</code>	A flag indicating whether a file has been marked read-only or locked by the system.
<code>FileReference.size</code>	The size of the file in bytes.
<code>FileReference.type</code>	The file extension (Windows) or Macintosh 4 character file type.

## Event handler summary for the FileReference object

Event handler	Description
<code>FileReference listener.onDownloadFailed</code>	Called when an error occurs while downloading.
<code>FileReference listener.onDownloadProgress</code>	Called while downloading, to report transfer progress.
<code>FileReference listener.onDownloadStart</code>	Called at the beginning of a download operation.
<code>FileReference listener.onDownloadSuccess</code>	Called when downloading completes successfully.
<code>FileReference listener.onFileError</code>	Called when an error occurs during a file operation.
<code>FileReference listener.onEndOfFile</code>	Called when the end of the file is reached during a read operation.
<code>FileReference listener.onUploadFailed</code>	Called when an error occurs while uploading.
<code>FileReference listener.onUploadProgress</code>	Called while uploading, to report transfer progress.
<code>FileReference listener.onUploadStart</code>	Called at the beginning of an upload operation.
<code>FileReference listener.onUploadSuccess</code>	Called when uploading completes successfully.

## Initializing FileReference objects

You can initialize FileReference objects in the following three ways:

- The `browse()` method allows the application to open any file on the user's disk. The user is prompted to choose a specific file.
- The `download()` method initiates the downloading of a remote URL, and prompts the users where to save the file.

- The `create()` and `open()` methods allow the application to open a file in the user's local Internet cache. This file may be deleted if the size of the local Internet cache exceeds the limit set by the user.

After a `FileReference` object has been initialized, it can be read, written, and uploaded to a remote URL using HTTP POST.

## Uploading and Downloading files

You can upload and download files using the `upload()` and `download()` methods. These methods initiate a file transfer and return immediately. You can track progress of the transfer using events. In order to receive events, the application should create a listener object that processes events as they happen.

**Note:** Some server-side upload scripts may limit the amount of POST data received using the upload method (for example, a .NET server will limit the POST data size to 4MB maximum unless you edit the web application's `web.config` file manually). Please consult your server development documentation for further details.

The following example uploads a file and keeps track of its progress:

```
var listener = new Object();

// show the upload progress to the user
listener.onUploadProgress = function(fileRef, sent, size) {
    gShell.setProgress(sent / size * 100);
    gShell.setStatus("Uploaded " + sent + " of " + size + " bytes");
}

// when the upload operation is complete, update the status and progress
listener.onUploadSuccess = function(fileRef, response) {
    gShell.setProgress(0);
    gShell.setStatus("File upload completed");
}

// the upload operation may fail
listener.onUploadFailed = function(fileRef, error, response) {
    gShell.setProgress(0);
    gShell.setStatus("File upload failed");
}

// initiate the upload operation with a pre-initialized FileReference object
function uploadFile(fileRef) {
    fileRef.addListener(listener);
    gShell.setStatus("Beginning to upload file");

    // initiate the upload operation
    fileRef.upload("http://www.mysite.com/cgi-bin/submit.cgi");
}
```



## Reading and Writing files

An application can read from a file with `readFile()`, `readBytes()` or `readString()`. An application can also write to a file using `writeBytes()` or `writeString()`. When file reading or writing is complete, the application should call `close()` to close the file. The file also closes when the `FileReference` object is destroyed.

When a file is read with `readFile()`, the entire contents of the file are read and returned in a string.

All other read and write methods maintain a current read/write position in the file. When a file is read or written with one of these methods, the read/write position moves forward by the number of bytes read. You can access the current read/write position with `getPosition()` and change it with `setPosition()`.

## Constructor for the FileReference object

### Availability

Macromedia Central 1.5

### Usage

```
fileRef = new FileReference();
```

### Parameters

None.

### Returns

Nothing.

### Description

Constructor; creates a `FileReference` object. You cannot use the `FileReference` object until you initialize it with `FileReference.browse()`, `FileReference.create()`, `FileReference.open()`, or `FileReference.download()`.

### Example

```
// first create the object
var fileRef = new FileReference();

// now initialize it with create()
fileRef.create("log.txt");
```

## FileReference.browse()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.browse( [ typelist [, defaultName] ] );
```

## Example

```
// ask the user to choose an image file for upload
var fileRef = new FileReference();
if (fileRef.browse(["Images", "*.jpg;*.gif;*.png", "Flash Movies", "*.swf"]))
{
    trace("Opened " + fileRef.name);
} else {
    trace("User cancelled");
}
```

## Parameters

*typelist*    Array of strings; describes the list of file types that the application can accept from the user.

*defaultName*    String; the default name of a file to request from the user.

## Returns

A Boolean value: `true` if the user chose a file; `false` if the user did not choose a file.

## Description

Method; initializes the `FileReference` object by prompting the user to select a file on the local disk. This method also grants permission to read the file. If the application performs any write operations, the user is prompted to confirm that write permission is allowed.

The *typelist* parameter is used to limit the user's selection in the file browse dialog box by presenting a list of one or more filters. Each filter is represented by a set of two or three strings, as follows:

- The first string is a user-visible description of the file, such as "Text Files".
- The second string is a semicolon-separated list of wildcard matches, such as "\*.txt;\*.ini".
- The third string is optional, and is a semicolon-separated list of Macintosh file types, such as "TEXT;ttxt". This parameter is ignored if Central is not running on the Macintosh platform.

For example, to open an image or Flash SWF file, you might pass the array `["Image files", "*.jpg;*.gif;*.png", "JPEG;jp2_;GIFf", "Flash Movies", "*.swf", "SWFL"]`. If you want to include an option for every file, you must manually include a wildcard entry, such as `"All files", "*.*"`.

If you omit the *typelist* parameter, the default filter includes all files, and any file is selectable by the user.

Use the *defaultName* parameter to suggest the name of a file to the user. This name will appear in the title bar of the file selection dialog.

You cannot use the `browse()` method to create new files. For information about creating files, see ["FileReference.create\(\)" on page 244](#).

## FileReference.close()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.close();
```

### Example

```
// read vars.txt if it exists
var fileRef = new FileReference();
if (fileRef.open("vars.txt")) {
    data = fileRef.readFile();
    fileRef.close();
}
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; closes a file that was previously opened by a call to `open()` or `create()`. The `close()` method clears all the properties on the object, and the reference to the file itself. Execute methods such as `saveAs()` or `copy()` before closing the object. The file also closes when the `FileReference` object goes away.

## FileReference.copy()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.copy(newName);
```

### Example

```
// make a backup copy of vars.txt if it exists
var fileRef = new FileReference();
if (fileRef.open("vars.txt")) {
    fileRef.copy("vars2.txt");
}
```

### Parameters

*newName* String; the relative name of the new file in the local Internet cache.

### Returns

A Boolean value that indicates the success or failure of the file copy operation. A `true` value indicates success.

## Description

Method; makes a copy of a file in the local Internet cache.

## FileReference.copyIntoCache()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.copyIntoCache( newName );
```

### Example

```
// make a working copy of a file on the user's disk
var fileRef = new FileReference();
if (fileRef.browse(["Text Files", "*.txt"])) {
    fileRef.copyIntoCache("backup-data.txt");
    var newFile = new FileReference();
    newFile.open("backup-data.txt");

    // operate on new file
    // ...

}
```

### Parameters

*newName*   String; the name of the new file.

### Returns

A Boolean value that indicates the success or failure of the file copy operation. A true value indicates success.

## Description

Method; creates a copy of a file and places it in the local Internet cache. The current FileReference object continues to refer to the original file.

## FileReference.create()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.create(relativePath [, bOverWrite]);
```

### Example

```
// create vars.txt and write one entry
var mapValue = "right at the light.";
var fileRef = new FileReference();

// create the file if necessary
if (!fileRef.exists("vars.txt")) {
```

```

        fileRef.create("vars.txt", false)
    } else {
        fileRef.open("vars.txt");
    }

    // now store the value
    fileRef.writeString("map=" + mapValue);

```

### Parameters

*relativePath* String; specifies a relative path to a file in the local Internet cache.

*boOverWrite* Boolean value; indicates whether the file should be overwritten if it already exists. If the parameter is set to `true` and the file already exists, a new empty file will overwrite the existing one. The default value is `false`.

### Returns

A Boolean value: `true` if the `create()` method succeeds. The function only returns `false` if the file was not created and the file did not previously exist (for example, in the case of an invalid filename). If this parameter is set to `false` and the file already exists, the `create()` method will return a value of `true`.

### Description

Method; creates a file in the user's local Internet cache. The file is stored relative to the calling SWF file. If the file exists and is overwritten by setting the `overwrite` parameter to `true`, the existing file is erased and the file has a size of zero.

If the `relativePath` parameter contains directory names, the directories are created as necessary. For example, if *relativePath* is "photos/january/home.jpg" and the photos directory does not exist, the photos directory and the january subdirectory are created.

The `FileReference` object has read, write, and upload permissions granted when it is created.

To create files outside of the local Internet cache, use the `create()` method to create the file in the cache, and then use the `saveAs()` method to prompt the user for a new location. For more information about using `saveAs()`, to change files outside of the local Internet cache, see [“FileReference.saveAs\(\)” on page 263](#).

## FileReference.creationDate

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.creationDate
```

### Example

```

// retrieve the creation date of a file chosen by the user
var fileRef = new FileReference();
if (fileRef.browse(["JPEG files", "*.jpg"])) {
    trace("File was created on " + fileRef.creationDate);
}

```

### Description

Date object; a read-only property that represents the date that the file was originally created.

## FileReference.creator

### Availability

Macromedia Central 1.5

### Usage

*fileRef.creator*

### Example

```
// retrieve the creator of a file chosen by the user
var fileRef = new FileReference();
if (fileRef.browse(["JPEG files", "*.jpg"])) {
    trace("File is of type " + fileRef.creator);
}
```

### Description

String; only available on the Macintosh. It is the four-character creator type of the file. If this property is set by the application, it changes the creator ID on the disk.

This property is always undefined on Windows.

## FileReference.deleteFile()

### Availability

Macromedia Central 1.5

### Usage

*fileRef.deleteFile();*

### Example

```
var fileRef = new FileReference();

if (fileRef.open("temp.txt")) {
    data = fileRef.readFile();
    fileRef.deleteFile();
}
```

### Parameters

None.

### Returns

A Boolean value: indicates the success or failure of the file deletion. A `true` value indicates that the file was deleted successfully.

### Description

Method; deletes a file in the local Internet cache that has been opened with write permission. Files cannot be deleted after they have been closed.

## FileReference.download()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.download(remoteURL)
```

### Example

```
var fileRef = new FileReference();
var listener = new Object();
listener.onDownloadSuccess = function(fileRef) {
    trace("Download complete.");
}

fileRef.addListener(listener);
if (fileRef.download("http://www.mysite.com/banner.gif")) {
    trace("Download has begun");
} else {
    trace("Download aborted");
}
```

### Parameters

*remoteURL*   String; the URL of a file to be downloaded.

### Returns

A Boolean value: `true` indicates that the user chose a destination for the downloaded file; otherwise `false`.

### Description

Method; initializes the `FileReference` by prompting the user for a location to store a downloaded file. It then begins downloading the file into that location. The call to the `download()` method returns as soon as the user has chosen a target location. If the user does not select a location for the file, usually by pressing the Cancel button in the file selection dialog box, the `download()` method returns `false`.

To track the progress and completion of the download operation, an application should use an event listener object to receive event notifications. The following notifications are sent during the download operation:

Event	Description
<code>FileReference</code> <code>listener.onDownloadStart</code>	The download has begun.
<code>FileReference</code> <code>listener.onDownloadProgress</code>	Some data has been downloaded.
<code>FileReference</code> <code>listener.onDownloadSuccess</code>	The download operation has finished and the file is saved to disk.
<code>FileReference</code> <code>listener.onDownloadFailed</code>	The download operation failed and the file has been deleted.

Applications can use the file after the `onDownloadSuccess` event has occurred. The `onDownloadProgress` events may occur before, between, or after `onDownloadStart` and `onDownloadSuccess` events. Applications should use the `onDownloadProgress` event solely to provide visual feedback to the user, and should not make any assumptions about the order of these events.

When the download operation is complete, the application does not have any permissions to read or write the file. If the application begins a read or write operation with a downloaded file, the user is prompted to grant permission for the given operation.

## FileReference.exists()

### Availability

Macromedia Central 1.5

### Usage

```
fileExists = FileReference.exists(relativePath);
```

### Example

```
var fileRef = new FileReference();

// read the file if it has not been locked by another
// application. Other applications will create a .lck
// file to indicate that the file is locked.
if (!fileRef.exists("data.lck")) {
    fileRef.open("data.txt");
    data = fileRef.readFile();
    fileRef.close();
}
```

### Parameters

*relativePath* String; the relative path to a file in the local Internet cache.

### Returns

A Boolean value that indicates whether the file exists.

### Description

Method; determines whether a file exists in the local Internet cache. This method is frequently used to determine if a file should be opened or created. The `FileReference` object does not need to be initialized to call the `exists()` method.

## FileReference.getPosition()

### Availability

Macromedia Central 1.5

### Usage

```
position = fileRef.getPosition();
```



### Example

```
// write an array of strings and keep track of total progress
function saveValues(strings) {
    var fileRef = new FileReference();
    fileRef.create("strings.txt");

    // write the strings to the file
    for (var i=0; i<strings.length; i++) {
        var data = fileRef.writeString(strings[i] + "\n");
        trace("Written " + fileRef.getPosition() + " bytes so far");
    }

    fileRef.close();
}
```

### Parameters

None.

### Returns

A number that indicates the current read and write position in the current file.

### Description

Method; returns the current read and write position in the current file. Future calls to `writeBytes()` and `writeString()` begin writing at this offset in the file. You can use this method to determine how many bytes have been read, or use it in conjunction with the `setPosition()` method as an aid in random access of file contents.

For more information about changing the current read and write location of a file, see [“FileReference.setPosition\(\)” on page 265](#).

## FileReference.locate()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.locate()
```

### Example

```
var fileRef = new FileReference();
fileRef.download("http://www.mysite.com/photos/photo.jpg");
fileRef.locate();
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; displays the location of the file to the user using the operating system's file manager. On Windows, a Windows Explorer window is opened for the containing folder. On the Macintosh, the Finder is used to display the folder that contains the file.

## FileReference.modificationDate

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.modificationDate
```

### Example

```
var fileRef = new FileReference();
if (fileRef.open("data.txt")) {
    trace("File was last modified: "+ fileRef.modificationDate);
}
```

### Description

Date object; a read-only property that represents the current date of the file as it is stored by the system.

## FileReference.move()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.move( [ otherFileRef ] )
```

### Example

```
function MoveDataTo(newFileRef) {
    var fileRef = new FileReference();

    // move the existing data.txt to a new location
    if (fileRef.open("data.txt")) {
        fileRef.move(newFileRef);
    }
}
```

### Parameters

*otherFileRef* FileReference; an optional destination for the file.

### Returns

A Boolean value; determines whether the move succeeded, or failed due to a file error or due to the user cancelling the operation.

## Description

Method; moves an opened file to a new location. If the *otherFileRef* parameter is omitted, the user is prompted for a destination for the file. If a *FileReference* object is supplied for the *otherFileRef* parameter, the user is prompted to confirm the destination of the file. For more information about changing files outside of the local Internet cache, see [“FileReference.saveAs\(\)” on page 263](#).

## FileReference.name

### Availability

Macromedia Central 1.5

### Usage

*fileRef.name*

### Example

```
var fileRef = new FileReference();

if (fileRef.browse(["Text Files", "*.txt"])) {
    trace("User chose the file "+ fileRef.name);
}
```

### Description

String; a read-only property that represents the name of the file on the local disk.

It does not include the path to the file and it cannot be used to rename the file.

## FileReference listener.onDownloadFailed

### Availability

Macromedia Central 1.5

### Usage

```
listener.onDownloadFailed = function(fileRef, error) {
}
```

### Example

```
var listener = new Object();
listener.onDownloadFailed = function(fileRef, error) {
    if (error == 404)
        trace("That URL does not exist");
    else
        trace("Error downloading " + fileRef.name);
}

var fileRef = new FileReference();
fileRef.addListener(listener);
fileRef.download("http://www.mysite.com/image1.jpg");
```

## Parameters

*fileRef* FileReference object; the object that initiated the event.

*error* Number; the network error number.

## Returns

Nothing.

## Description

The `onDownloadFailed` event is called when there is an error in downloading the file. This can be caused by a bad URL or a broken connection during the download operation.

The *error* parameter may be -1 for a general network error, or an HTTP error number, such as 404, to indicate that a page is not found. HTTP error values can be found in sections 10.4 and 10.5 of the HTTP specification at <ftp://ftp.isi.edu/in-notes/rfc2616.txt>.

See also “[FileReference.download\(\)](#)” on page 247.

## FileReference listener.onDownloadProgress

### Availability

Macromedia Central 1.5

### Usage

```
listener.onDownloadProgress = function(fileRef, received, size) {  
    }  
}
```

### Example

```
var listener = new Object();  
listener.onDownloadProgress = function(fileRef, received, size) {  
    trace("Received " + (received/size*100) + "% of file");  
}  
  
var fileRef = new FileReference();  
fileRef.addListener(listener);  
fileRef.download("http://www.mysite.com/image1.jpg");
```

## Parameters

*fileRef* FileReference object; the object that initiated the event.

*received* Number; the total bytes downloaded so far.

*size* Number; the total number of bytes to be downloaded. The *size* parameter is -1 if the final size cannot be determined.

## Returns

Nothing.

## Description

Event; called periodically during a file download operation. It reflects the number of bytes already downloaded, as well as the total bytes to be downloaded. The total download size cannot always be determined until the download is complete. In this case, the *size* parameter is -1 for all `onDownloadProgress` events.

The *received* parameter refers to the number of bytes transferred, including any protocol headers. It does not refer to the current total size of the completed file. Applications should not depend on the *size of received* parameters, except to provide visual feedback for download progress.

See also [“FileReference.download\(\)” on page 247](#).

## FileReference listener.onDownloadStart

### Availability

Macromedia Central 1.5

### Usage

```
listener.onDownloadStart = function(fileRef) {  
    }  
}
```

### Example

```
var listener = new Object();  
listener.onDownloadStart = function(fileRef) {  
    trace("Download of " + fileRef.name + " has begun");  
}  
  
var fileRef = new FileReference();  
fileRef.addListener(listener);  
fileRef.download("http://www.mysite.com/image1.jpg");
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

### Returns

Nothing.

## Description

Event; called when a file download operation is initiated by a call to [FileReference.download\(\)](#). This event only indicates that a file download has been requested, but does not indicate any success or failure of the initial network connection.

See also [“FileReference.download\(\)” on page 247](#) and [“FileReference listener.onDownloadFailed” on page 251](#).

## FileReference listener.onDownloadSuccess

### Availability

Macromedia Central 1.5

### Usage

```
listener.onDownloadSuccess = function(fileRef) {  
    }  
}
```

### Example

```
var listener = new Object();  
listener.onDownloadSuccess = function(fileRef) {  
    trace("Download of " + fileRef.name + " complete.");  
}  
  
var fileRef = new FileReference();  
fileRef.addListener(listener);  
fileRef.download("http://www.mysite.com/image1.jpg");
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

### Returns

Nothing.

### Description

Event; called when a file download operation has successfully completed. After this event has been called, it is safe to use the file.

See also [“FileReference.download\(\)” on page 247](#) and [“FileReference listener.onDownloadFailed” on page 251](#).

## FileReference listener.onFileError

### Availability

Macromedia Central 1.5

### Usage

```
listener.onFileError = function(fileRef, errorCode, errorText) {  
    }  
}
```

### Example

```
var listener = new Object();  
listener.onFileError = function(fileRef, errorCode, errorText) {  
    trace("Error number " + errorCode + " reading " + fileRef.name + ": " +  
        errorText);  
}  
  
var fileRef = new FileReference();  
fileRef.addListener(listener);
```

```
// handle errors gracefully, let the listener deal with specifics
if (fileRef.open("file.txt")) {
    fileRef.upload("http://www.mysite.com/upload.cgi");
}
```

## Parameters

- fileRef*    **FileReference** object; the object that initiated the event.
- errorCode*    Number; identifies the particular error.
- errorText*    String; a text description of the error that can be presented to the user.

## Returns

Nothing.

## Description

Event; called when a general file error occurs. This may occur during reading, writing, opening, or closing a file. The *errorCode* parameter identifies the particular error, and the *errorText* parameter provides a verbose description that can be presented to the user.

Most **FileReference** methods indicate success or failure of the operation. The **onFileError** event is provided to determine what caused the error.

The following is a list of errors and descriptions:

Error code	Error text	Description
-1	none	A generic error has occurred and Central cannot identify its cause.
-2	Access Denied	An error has occurred writing, renaming, or deleting a file that is read-only or already opened by another application.
-3	File Not Found	The target file cannot be found.
-4	Security Violation	The requested action is disallowed by system policy; for example, attempting to write to an executable file.
-5	Invalid File Name	The creation or renaming of a file failed because the filename is not valid.
-6	Out of Disk Space	The disk is full and no more data can be written to this file.
-7	User Denied Access	The user is denied permission to complete the action.
-8	File Limit Exceeded	The maximum file size of 20 megabytes has been reached.

## FileReference listener.onEndOfFile

### Availability

Macromedia Central 1.5

### Usage

```
listener.onEndOfFile = function(fileRef) {
```

```
}
```

### Example

```
var listener = new Object();
listener.onEndOfFile = function(fileRef) {
    trace("Ran out of data reading "+ fileRef.name);
}

var fileRef = new FileReference();
fileRef.addListener(listener);

// handle errors gracefully, let the listener deal with specifics
if (fileRef.open("file.txt")) {
    str = fileRef.readString(100);
}
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

### Returns

Nothing.

### Description

Event; occurs when a read operation reaches the end of a file during the read operation.

See also [“FileReference.readBytes\(\)” on page 260](#), [“FileReference.readFile\(\)” on page 261](#), and [“FileReference.readString\(\)” on page 262](#).

## FileReference listener.onUploadFailed

### Availability

Macromedia Central 1.5

### Usage

```
listener.onUploadFailed = function(fileRef, error, response) {
}
```

### Example

```
var listener = new Object();
listener.onUploadFailed = function(fileRef, err, response) {
    trace("Error number " + err + " during upload");
}

var fileRef = new FileReference();
fileRef.addListener(listener);

// create a file and fill it with data
fileRef.create("data.txt");
fileRef.writeString("enable=true\n");
```



```
// upload the file
fileRef.upload("http://www.mysite.com/upload.cgi");
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

*error* Number; the network error number.

*response* String; the body of the response from the server.

### Returns

Nothing.

### Description

Event; called when there is an error uploading a file. This could occur because of a bad URL, because the server rejected the HTTP POST, or because of an error during the transfer.

The *error* parameter may be -1 for a general network error, or an HTTP error number, such as 404, to indicate that a page is not found. HTTP error values can be found in sections 10.4 and 10.5 of the HTTP specification at <ftp://ftp.isi.edu/in-notes/rfc2616.txt>.

The *response* parameter can be used to capture the body of the HTTP response from the server.

See also [“FileReference.upload\(\)” on page 266](#).

## FileReference listener.onUploadProgress

### Availability

Macromedia Central 1.5

### Usage

```
listener.onUploadProgress = function(fileRef, sent, size) {
}
```

### Example

```
var listener = new Object();
listener.onUploadProgress = function(fileRef, sent, size) {
    trace("Uploaded " + sent + " of " + size + " bytes");
}

var fileRef = new FileReference();
fileRef.addListener(listener);

// create a file and fill it with data
fileRef.create("data.txt");
fileRef.writeString("enable=true\n");

// upload the file
fileRef.upload("http://www.mysite.com/upload.cgi");
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

*sent* Number; the number of bytes uploaded so far.

*size* Number; the total size of the file to be uploaded, in bytes.

### Returns

Nothing.

### Description

Event; called periodically during a file upload operation. It reflects the number of bytes already uploaded, as well as the total bytes to be uploaded.

The *sent* parameter refers to the number of bytes transferred, including any protocol headers. It does not refer to the size of the file. The *size* parameter refers to the total size of the file to be uploaded. Applications should not depend on the *size* or *sent* parameters, except to provide visual feedback for upload progress. The [FileReference](#) `listener.onUploadSuccess` event indicates that the file transfer has finished successfully.

On Macintosh systems before OS X 10.3, the progress of the upload operation cannot be determined. The `onUploadProgress` event is called during the upload operation, but the *sent* parameter will be -1 to indicate an indeterminant progress.

See also “[FileReference.upload\(\)](#)” on page 266.

## FileReference listener.onUploadStart

### Availability

Macromedia Central 1.5

### Usage

```
listener.onUploadStart = function(fileRef) {  
    }  
}
```

### Example

```
var listener = new Object();  
listener.onUploadStart = function(fileRef) {  
    trace("Upload has begun");  
}  
  
var fileRef = new FileReference();  
  
// ask the user to choose a file to upload  
if (fileRef.browse(["Text Files", "*.txt"])) {  
    fileRef.addListener(listener);  
    fileRef.upload("http://www.mysite.com/upload.cgi");  
}
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

### Returns

Nothing.

## Description

Event; called when a file upload operation is initiated by a call to [FileReference.upload\(\)](#). This event only indicates that a file upload operation has been requested, but does not indicate the success or failure of the initial network connection.

See also [“FileReference.upload\(\)” on page 266](#) and [“FileReference listener.onUploadFailed” on page 256](#).

## FileReference listener.onUploadSuccess

### Availability

Macromedia Central 1.5

### Usage

```
listener.onUploadSuccess = function(fileRef, response) {  
}
```

### Example

```
var listener = new Object();  
listener.onUploadSuccess = function(fileRef) {  
    trace("Upload complete");  
}  
  
var fileRef = new FileReference();  
  
// ask the user to choose a file to upload  
if (fileRef.browse(["Text Files", "*.txt"])) {  
    fileRef.addListener(listener);  
    fileRef.upload("http://www.mysite.com/upload.cgi");  
}
```

### Parameters

*fileRef* FileReference object; the object that initiated the event.

*response* String; the body of the response from the server.

### Returns

Nothing.

### Description

Event; called when a file upload operation has completed successfully. The *response* parameter can be used to capture the HTTP output from the server.

See also [“FileReference.upload\(\)” on page 266](#)

## FileReference.open()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.open(relativePath);
```

### Example

```
var fileRef = new FileReference();

// open the file and read 100 bytes
if (fileRef.open("data.txt")) {
    data = fileRef.readString(100);
} else {
    trace("Cannot open data.txt");
}
```

### Parameters

*relativePath* String; the relative path to an existing file in the local Internet cache.

### Returns

A Boolean value that indicates whether the file was opened successfully.

### Description

Method; opens an existing file in the local Internet cache.

The FileReference object has read, write, and upload permissions granted when the file is opened.

## FileReference.readBytes()

### Availability

Macromedia Central 1.5

### Usage

```
data = fileRef.readBytes(count);
```

### Example

```
var fileRef = new FileReference();

if (fileRef.open("data.dat")) {
    // read a 3 byte header from the file
    var data = fileRef.readBytes(3);
    if (data[0] != 25 || data[1] != 100 || data[2] != 255) {
        trace("Invalid signature, file is corrupt");
    }
}
```

### Parameters

*count* Number; the number of bytes to read.

### Returns

An array of numbers that contains the next *count* bytes in the file.

## Description

Method; reads the specified number of bytes at the current read-write position, and returns an Array of integers, with values ranging from 0 to 255. This method is commonly used to read binary data from a file.

After the data has been read, the read-write position is moved forward by the specified number of bytes.

The length of the resulting Array indicates the number of bytes read. If the end of the file is reached before the requested number of bytes are read, the resulting Array contains only the remaining bytes in the file.

**Note:** The recommended limit on the fileReference object for the `readBytes()` method is 1MB, for performance. However, Central does not enforce an absolute limit.

See also [“FileReference.readFile\(\)” on page 261](#) and [“FileReference.setPosition\(\)” on page 265](#).

## FileReference.readFile()

### Availability

Macromedia Central 1.5

### Usage

```
string = fileRef.readFile();
```

### Example

```
// open data file and read into file_data
var fileRef = new FileReference();
if (fileRef.open("data.txt")) {
    file_data = fileRef.readFile();
}
```

### Parameters

None.

### Returns

A string that contains the contents of the entire file.

## Description

Method; reads the entire file contents into a string. It reads from the beginning of the file, ignoring any previous calls to [FileReference.setPosition\(\)](#). This method should only be used to read text files.

After the data has been read, the read-write position is moved to the end of the file.

See also [“FileReference.readBytes\(\)” on page 260](#) and [“FileReference.readString\(\)” on page 262](#).

## FileReference.readOnly

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.readOnly
```

### Example

```
var fileRef = new FileReference();
if (fileRef.browse(["Text files", "*.txt"])) {
    if (fileRef.readOnly)
        trace("File is read-only");
}
```

### Description

String; a read-only Boolean property that indicates whether the system has locked the file or marked it read-only. If it is `true`, the file is locked and can only be read or uploaded.

This property is not related to the file permissions granted to the FileReference object. For example, it is possible for a user to choose a locked file in response to a call to [FileReference.browse\(\)](#) with the “w” write property requested. In this case, the `readOnly` property is `true`.

The property itself is read-only and cannot be changed.

## FileReference.readString()

### Availability

Macromedia Central 1.5

### Usage

```
string = fileRef.readString(count);
```

### Example

```
var fileRef = new FileReference();
if (fileRef.open("local.txt")) {

    // make sure the file begins with "FILESTART0"
    data = readString(10);
    if (data != "FILESTART9") {
        trace("Invalid signature, file is corrupt!");
    }
}
```

### Parameters

*count*    Number; the number of bytes to read.

### Returns

A string that contains the next *count* bytes in the file.

## Description

Method; reads data from the file at the current read-write position. After the data has been read, the read-write position is moved forward. This method should only be used to read text files.

The length of the resulting string indicates the number of bytes read. If the end of the file is reached before the requested number of bytes are read, the resulting string contains only the remaining bytes in the file.

See also [“FileReference.readBytes\(\)” on page 260](#), [“FileReference.readFile\(\)” on page 261](#), and [“FileReference.setPosition\(\)” on page 265](#).

## FileReference.rename()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.rename( newName );
```

### Example

```
var fileRef = new FileReference();
if (fileRef.open("beach.jpg") {
    fileRef.rename("beach_backup.jpg");
}
```

### Parameters

*newName* String; the new name of the file.

### Returns

Nothing.

### Description

Method; changes the name of an existing file in the local Internet cache. The file must have write permission to call this method.

For more information about moving or renaming files that exist outside of the local Internet cache, see [“FileReference.saveAs\(\)” on page 263](#) and [“FileReference.move\(\)” on page 250](#).

## FileReference.saveAs()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.saveAs( [ otherFileRef ] );
```

### Example

```
var fileRef = new FileReference();

// create a temporary version of the file
```

```
fileRef.create("vars.txt");
fileRef.writeString("lastDocument=spinach.txt");

// prompt the user to save the file
fileRef.saveAs();
```

## Parameters

*otherFileRef* **FileReference**; an optional destination for the new file.

## Returns

A Boolean value; indicates whether the `saveAs` succeeded, or failed due to a file error or the user cancelling the operation.

## Description

Method; makes a copy of a file and saves it to a new location. If the *otherFileRef* parameter is omitted, Central prompts the user for a destination for the new file. If the *otherFileRef* parameter is included, Central prompts the user for permission to save the file in the new location.

This method can be used to create or overwrite files outside of the local Internet cache. The following example demonstrates how to modify a file outside of the cache.

```
var fileRef1 = new FileReference();
if (fileRef1.browse())
{
    // the local name of the file in the local Internet cache
    var localName = fileRef1.name;

    // make a copy in the cache
    fileRef1.copyIntoCache(localName);

    // open the newly copied file
    var fileRef2 = new FileReference();
    fileRef2.open(localName);

    // write some data
    fileRef2.writeString("buildID=14");

    // now copy the updated file back to
    // the original location outside the cache
    fileRef2.saveAs(fileRef1)
}
```

For more information about changing files outside of the local Internet cache, see [“FileReference.copyIntoCache\(\)” on page 244](#).



## FileReference.setPosition()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.setPosition(newPosition);
```

### Example

```
// read the specified record number and return it in a byte array
function readRecord(fileRef, recordNumber) {
    if (fileRef.setPosition(recordNumber * gRecordSize))
        return fileRef.readBytes(gRecordSize);
    return null;
}
```

### Parameters

*newPosition*    Number; the current read or write offset from the beginning of the file, in bytes.

### Returns

A Boolean value: `true` indicates that the `setPosition()` method was successful; otherwise `false`.

### Description

Method; sets the current read or write position in a file. It is commonly used to provide random access to a binary file.

The *newPosition* parameter is always an absolute value from the beginning of the file. To set relative position, use the `getPosition()` method in conjunction with the `setPosition()` method.

For example, the following code rewinds 10 bytes in a file:

```
fileRef.setPosition(fileRef.getPosition() - 10);
```

The `setPosition()` method affects future calls to `readBytes()`, `readString()`, `writeBytes()`, and `writeString()`.

## FileReference.size

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.size
```

### Example

```
// retrieve the size of a file chosen by the user
var fileRef = new FileReference();
if (fileRef.browse(["JPEG files", "*.jpg"])) {
    trace("File is " + fileRef.size + " bytes");
}
```

### Description

Number; a property that indicates the size of the file, in bytes.

## FileReference.type

### Availability

Macromedia Central 1.5

### Usage

*fileRef.type*

### Description

String; a property that represents the file extension of the filename on Windows, and the four-character file type on the Macintosh. This property is read-only on Windows. It is writable on the Macintosh and can be used to change the file type of a file.

## FileReference.upload()

### Availability

Macromedia Central 1.5

### Usage

*fileRef.upload(remoteURL);*

### Example

```
var listener = new Object();
listener.onUploadSuccess = function(fileRef, response) {
    trace("Upload of "+ fileRef.name + "complete");
    trace("Server response was "+ response);
}

var fileRef = new FileReference();

// ask the user to choose a file to upload
if (fileRef.browse(["Text Files", "*.txt"])) {
    fileRef.addListener(listener);
    fileRef.upload("http://www.mysite.com/upload.cgi");
}
```

### Parameters

*remoteURL* String; specifies the URL that will receive an HTTP POST of the file.

### Returns

Nothing.

### Description

Method; uploads a file to a remote server. The FileReference object must be initialized before the call to the `upload()` method. The call to the `upload()` method returns immediately.

To track the progress and completion of the upload operation, an application should use an event listener object to receive event notifications. The following notifications are sent during the upload operation:

Event	Description
<a href="#">FileReference</a> <code>listener.onUploadStart</code>	The upload operation has begun.
<a href="#">FileReference</a> <code>listener.onUploadProgress</code>	Some data has been uploaded.
<a href="#">FileReference</a> <code>listener.onUploadSuccess</code>	The upload operation has finished without an error.
<a href="#">FileReference</a> <code>listener.onUploadFailed</code>	The upload operation failed.

Applications should use the `onUploadProgress` event solely to provide visual feedback to the user, and should not make any assumptions about the order of these events. The `onUploadProgress` events may occur before, between, or after `onUploadStart` and `onUploadSuccess` events.

## FileReference.writeBytes()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.writeBytes(byteArray);
```

### Example

```
// data is an array of binary data
var data = new Array(25, 200, 255);

// create and initialize the FileReference
var fileRef = new FileReference();
fileRef.create("data.dat");

// write the data to disk
fileRef.writeBytes(data);
```

### Parameters

*byteArray* Array of numbers; represents the byte value to be written.

### Returns

A Boolean value that indicates the success or failure of the operation. The value `true` means that the write operation succeeded.

### Description

Method; writes an array of Numbers to the file. The values in the array must range from 0 to 255. This method is commonly used to write binary data to disk to later be read by [FileReference.readBytes\(\)](#).

See also “[FileReference.writeString\(\)](#)” on page 268.

## FileReference.writeString()

### Availability

Macromedia Central 1.5

### Usage

```
fileRef.writeString(str);
```

### Example

```
var fileRef = new FileReference();

// write data to a local file
fileRef.create("data.txt");
fileRef.writeString("enable=true\n");
```

### Parameters

*str* String; a string to write to the file.

### Returns

A Boolean value that indicates the success or failure of the operation. The value `true` means that the write operation succeeded.

### Description

Method; writes a string to the file. It does not write any line terminators, so the application must include line terminators if it is writing to a text file.

See also “[FileReference.writeBytes\(\)](#)” on page 267.

## FileReferenceList object

**ActionScript Class Name** FileReferenceList

FileReferenceList is a container class that allows the user to select multiple files from the local disk. The [FileReferenceList.browse\(\)](#) method is used to prompt the user to select a set of files. These files are accessible through the [FileReferenceList.fileList](#) property.

### Method summary for the FileReferenceList object

Method	Description
<a href="#">FileReferenceList.browse()</a>	Prompts the user to browse for a set of files.

### Property summary for the FileReferenceList object

Property	Description
<a href="#">FileReferenceList.fileList</a>	An array of the files chosen by the user.

## Event handler summary for the FileReferenceList object

---

Event handler	Description
None.	

---

## Constructor for the FileReferenceList object

### Availability

Macromedia Central 1.5

### Usage

```
fileRefList = new FileReferenceList();
```

### Example

```
var fileRefList = new FileReferenceList();
if (fileRefList.browse(["Images", "*.jpg;*.gif;*.png", "Flash Movies",
    "*.swf"])) {
    trace("User chose " + fileRefList.fileList.length + "files");
} else {
    trace("User cancelled");
}
```

### Parameters

None.

### Returns

A FileReferenceList object.

### Description

Constructor; creates the FileReferenceList object. The FileReferenceList object cannot be used until [FileReferenceList.browse\(\)](#) is called to populate the `fileList` property.

## FileReferenceList.browse()

### Availability

Macromedia Central 1.5

### Usage

```
fileRefList.browse( [ typelist ] );
```

### Example

```
var fileRefList = new FileReferenceList();
if (fileRefList.browse(["Images", "*.jpg;*.gif;*.png", "Flash Movies",
    "*.swf"])) {
    for (var i=0; i<fileRefList.fileList.length; i++) {
        trace("Selected " + fileRefList.fileList[i].name);
    }
} else {
    trace("User cancelled");
}
```

## Parameters

*typelist* Array of strings; describes the list of file types that the application can accept from the user.

## Returns

A Boolean value; `true` if the user chose at least one file; `false` if the user did not choose any files.

## Description

Method; prompts the user to choose one or more files. For more information about the *typelist* parameter, see [“FileReference.browse\(\)” on page 241](#). When the user has chosen a set of files, the `FileReferenceList.fileList` property holds those files.

## FileReferenceList.fileList

### Availability

Macromedia Central 1.5

### Usage

```
files = FileReferenceList.fileList
```

### Example

```
var fileRefList = new FileReferenceList;
if (fileRefList.browse(["Text files", "*.txt"])) {

    // upload each file
    for (var i=0; i<fileRefList.fileList.length; i++) {
        var fileRef = fileRefList.fileList[i];

        // use the same listener object on each file
        fileRef.addListener(uploadListener);

        // now submit the file
        fileRef.upload("http://www.mysite.com/submit-files.cgi");
    }
}
```

### Description

Array; a property that contains an array of `FileReference` objects that the user has chosen using [FileReferenceList.browse\(\)](#).

# LCDataProvider object

**ActionScript Class Name** mx.central.data.LCDataProvider

LCDataProvider is an implementation of the DataProviderClass object. Traditionally, you create a DataProviderClass (containing the source data) that gets linked to a component instance onscreen (such as the DataGrid or List). When the data changes, the component instance reflects the changes. The LCDataProvider is used for the same purpose, except that the components can reside in your application or any of your pods, while the data remains in one place, such as in an agent. Once you set up the structure, the native Central ActionScript handles all the work through LocalConnections.

**Note:** This data provider is different from the regular DataProvider with Flash MX 2004 and previous versions of the Macromedia developer resource kits (DRKs).

Like the LCService object, LCDataProvider objects use a client/server metaphor. The data resides in the Server side LCDataProvider instance (which should be implemented by your agent) and each Client LCDataProvider instance (in your application and pods) automatically communicate with the Server side. Changes you make to the data automatically propagate through the server and to all LCDataProvider Clients.

Creating the client and server parts of an LCDataProvider object is similar to the [LCService object](#), but does not require the developer to specify an interface. (An interface is a list of supported methods.) The interface is predefined, as it supports all standard DataProviderClass functions (except custom sorts using `sort(compareFunction)`, which is not supported).

The LCDataProvider Server object has the additional `setData()` function that you use to populate the DataProviderClass with an array or with another DataProvider.

The Central implementation of the LocalConnection object (LCService and LCDataProvider) executes calls synchronously. This means that you can call a Client LCDataProvider `getItemAt()` and get the result returned immediately, just like a normal DataProvider.

When an application's `onDeactivate()` method is called, all open LCDataProvider objects should be destroyed by calling `delete`.

## Method summary for the LCDataProvider object

The following table summarizes the methods for the LCDataProvider object:

Method	Description
<code>LCDataProvider.addItem()</code>	Adds a single item, <i>item</i> , to the end of the list of items.
<code>LCDataProvider.addItemAt()</code>	Adds a single item at a specific index in the list of items.
<code>LCDataProvider.addItems()</code>	Adds a set of item objects to the end of the list of items. This set can be either an array of objects (ordered) or an object of objects (unordered).
<code>LCDataProvider.addItemsAt()</code>	Adds a set of item objects to a specific index in the list of items.

Method	Description
<code>LCDDataProvider.addListener()</code>	Passes a reference to an object that is added to the <i>LCDDataProvider._listeners</i> array, and determines whether to call the object's <i>modelChanged</i> event.
<code>LCDDataProvider.close()</code>	Frees the communication resources associated with an LCDDataProvider object.
<code>LCDDataProvider.createClient()</code>	Creates an instance of the client side version of an LCDDataProvider. The name you supply needs to match the name used by your agent when it called <i>createServer()</i> .
<code>LCDDataProvider.createServer()</code>	Creates an instance of a named LCDDataProvider to which components in your application and pods can subscribe.
<code>LCDDataProvider.getAllItems()</code>	Returns the entire items array. This method returns only a reference to the actual <i>LCDDataProviderInstance.items</i> array.
<code>LCDDataProvider.getIndexByKey()</code>	Returns the index of the first item (starting from the beginning of the list) whose property indicated by <i>key</i> matches the value passed in.
<code>LCDDataProvider.getIndicesByKey()</code>	Returns an array of indexes for items whose property specified by <i>key</i> matches <i>value</i> .
<code>LCDDataProvider.getItemAt()</code>	Returns the item at <i>index</i> .
<code>LCDDataProvider.getItemByKey()</code>	Returns the first item object found whose property specified in <i>key</i> matches <i>value</i> .
<code>LCDDataProvider.getItemID()</code>	Returns the value of the <i>item._ID_</i> property found at <i>index</i> .
<code>LCDDataProvider.getItemsByKey()</code>	Returns an array of objects whose property specified in <i>key</i> matches <i>value</i> .
<code>LCDDataProvider.getLength()</code>	Returns the total number of items in the data provider.
<code>LCDDataProvider.getSortState()</code>	Returns a sort object with two string properties: <i>sortField</i> and <i>order</i> .
<code>LCDDataProvider listener.modelChanged()</code>	Broadcasts when the data provider changes
<code>LCDDataProvider.removeAll()</code>	Removes all items in the data provider.
<code>LCDDataProvider.removeItemAt()</code>	Removes the item at <i>index</i> and returns it.
<code>LCDDataProvider.removeListener()</code>	Searches the <i>_listeners</i> array for the reference passed in, and removes it if found.
<code>LCDDataProvider.replaceAllItems()</code>	Deletes and replaces all of the items in the LCDDataProvider instance.
<code>LCDDataProvider.replaceItemAt()</code>	Overwrites an item at the specified index with a new item object.



Method	Description
<code>LCDDataProvider.setData()</code>	Used by either the client or server to identify the source data. Replace "LCDDataProvider" with the name of the variable you're using to hold the instance of the LCDDataProvider ( <code>myAlbums</code> in the examples above).
<code>LCDDataProvider.setItemByKey()</code>	Overwrites an existing item object using the key of any existing item property.
<code>LCDDataProvider.sort()</code>	Sorts the items using a custom function, similar to <code>Array.sort</code> , but lets you pass in additional arguments to persist and retrieve user-driven sort selections.
<code>LCDDataProvider.sortItemsBy()</code>	Sorts the items in the specified order, using the built-in <code>Array.sortOn</code> method with the item property specified in <i>key</i> .
<code>LCDDataProvider.updateItem()</code>	Overwrites an existing item object with a new item object.
<code>LCDDataProvider.updateItemByIndex()</code>	Overwrites an existing item object at <i>index</i> with a new item object.
<code>LCDDataProvider.updateView()</code>	Broadcasts events, other than the <code>modelChanged</code> event, for any objects that extend the LCDDataProvider object. This method is called for each <code>_listener</code> that is added when the LCDDataProvider instance triggers an event.

## Events for the LCDDataProvider object

By extending the LCDDataProvider object and overriding the `updateView` method, you can make the data provider broadcast to any method you define. All components listen for `modelChanged` events, so you must broadcast to the `modelChanged` event, or your implementation will not work with existing components. To broadcast to a different method, pass in an `eventObject` as described next for the `modelChanged` event:

Event	Description
<code>DataProviderClass listener.modelChanged()</code>	Broadcasts when the data provider changes.

### Example

The server side (your agent) is where you name and structure the source LCDDataProvider and, optionally, populate it with initial data. The following is an example from the agent:

```
onActivate=function(agentManager, agentID)
{
    //create an LCDDataProvider instance to store in the myDP variable
    this.myAlbums = mx.central.data.LCDDataProvider.createServer("my_albums");
    var albums = [ "Debut", "Greatest Hits", "Features"];
    this.myAlbums.setData( albums );
}
```

```

onDeactivate=function()
    // clean up server connection
    delete this.myAlbums;
};

```

The client side (pod or application) needs to create an `LCDataProvider` instance by passing both a unique string identifier for itself (in this case, by combining the second and third parameters received in `onActivate()`), as well as the same name the server used when creating itself ( in this case, "my\_albums"). The following code shows how the client can then link components to the data, as well as make changes to the source data:

```

onActivate = function(shell, appID, shellID)
{
    this.myID = appID+"_"+shellID;
    myAlbums = mx.central.data.LCDataProvider.createClient(this.myID,
        "my_albums");
    myComboBox.dataProvider = myAlbums;
    sListBox.dataProvider = myAlbums;

    sMyAddButton.onRelease = onMyAddPress;
};

onDeactivate = function()
{
    // clean up client connection
    delete this.myAlbums;
};

onMyAddPress = function()
{
    myAlbums.addItem(this.myInputField_txt.text);
};

```

## LCDataProvider.addItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItem(item)
```

### Parameters

*item* An object; the item to add.

### Returns

Nothing.

## Description

Method; adds a single item, *item*, to the end of the list of items. If the item has a property named `_ID_`, that property is overwritten with a new, unique value.

## Example

The following example creates an `LCDataProvider` client object, populates it with 6 initial items, and sets the `dataProvider` property of two components to `LCDataProvider` client object:

```
var myID:String;
var myDP:mx.central.data.LCDataProvider;
var myDataGrid:mx.controls.DataGrid;
var myList:mx.controls.List;

function onActivate(shell:mx.central.Shell, appID:Number, shellID:Number,
    baseTabIndex:Number, initiaData:Object):Void
{
    this.myID = appID+"_"+shellID;
    this.myDP = mx.central.data.LCDataProvider.createClient(this.myID, "my_dp");

    setData();

    this.myDataGrid.dataProvider = myDP;
    this.myList.dataProvider = myDP;
}

function setData(Void):Void {
    // adds 6 initial items
    for (var i=0; i<6; i++) {
        this.myDP.addItem( {label:"Item "+i, data:i});
    }
}
```

## LCDataProvider.addItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItemAt(index, item)
```

### Parameters

*index* The index at which to add the item.

*item* The item to add.

### Returns

Nothing.

### Description

Method; adds a single item at a specific index in the list of items. If the item has a property named `_ID_`, that property is overwritten with a new, unique value.

### Example

The following example adds an item to the data provider `myDP` at the fourth position:

```
myDP.addItemAt(3, {label:"this is the fourth Item", data:4});
```

## LCDataProvider.addItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItem(items)
```

### Parameters

*items* An array of objects (ordered) or an object of objects (unordered).

### Returns

Nothing.

### Description

Method; adds a set of item objects to the end of the list of items. This set can be either an array of objects (ordered) or an object of objects (unordered). Either way, the items are stored in a specific order once added. If any item has a property named `_ID_`, that property is overwritten with a new, unique value.

### Example

The following example creates an `LCDataProvider` client object, populates it with 3 initial items from an array, and sets the `dataProvider` property of two components to `LCDataProvider` client object:

```
var myID:String;
var myDP:mx.central.data.LCDataProvider;
var myDataGrid:mx.controls.DataGrid;
var myList:mx.controls.List;

function onActivate(shell:mx.central.Shell, appID:Number, shellID:Number,
    baseTabIndex:Number, initiaData:Object):Void
{
    this.myID = appID+"_"+shellID;
    this.myDP = mx.central.data.LCDataProvider.createClient(this.myID, "my_dp");

    setDataFromArray();
```

```

        this.myDataGrid.dataProvider = myDP;
        this.myList.dataProvider = myDP;
    }

    function setDataFromArray(Void):Void {
        // adds 3 initial items
        var aItems:Array = [ { label:"Item 1", data:1 },
                             { label:"Item 2", data:2 },
                             { label:"Item 3", data:3 } ];
        this.myDP.addItem(aItems);
    }

```

## LCDataProvider.addItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addItemAt(index, items)
```

### Parameters

*index*    The number at which to add the items.

*items*    An array of objects (ordered) or an object of objects (unordered).

### Returns

Nothing.

### Description

Method; adds a set of item objects to a specific index in the list of items. If any item has a property named `_ID_`, that property is overwritten with a new, unique value.

### Example

The following code provides an example of how to populate a data provider and use the `addItemAt()` method:

```

var myID:String;
var myDP:mx.central.data.LCDataProvider;
var myDataGrid:mx.controls.DataGrid;
var myList:mx.controls.List;

function onActivate(shell:mx.central.Shell, appID:Number, shellID:Number,
                    baseTabIndex:Number, initiaData:Object):Void
{
    this.myID = appID+"_"+shellID;
    this.myDP = mx.central.data.LCDataProvider.createClient(this.myID, "my_dp");

    setData();
}

```

```

        this.myDataGrid.dataProvider = myDP;
        this.myList.dataProvider = myDP;

        addDataFromArray();
    }

    function setData(Void):Void {
        // adds 6 initial items
        for (var i=0; i<6; i++) {
            this.myDP.addItem( {label:"Item "+i, data:i});
        }
    }
    // adds three more items to the list
    function addDataFromArray(Void):Void {
        var aItems:Array = [ { label:"Item 6", data:6 },
            { label:"Item 7", data:7 },
            { label:"Item 8", data:8 } ];
        this.myDP.addItemAt(6, aItems);
    }
}

```

## LCDataProvider.addListener()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
addListener(view [,doNotUpdate])
```

### Parameters

*view*    A reference to an object.

*doNotUpdate*    A Boolean value. If true, the item's `modelChanged` event is not triggered. If null, undefined, or false, a `modelChanged` event is triggered on the referenced object.

### Returns

Nothing.

### Description

Method; passes a reference to an object that is added to the `LCDataProvider._listeners` array, and determines whether to call the object's `modelChanged` event.

If *doNotUpdate* is true, the item's `modelChanged` event is not triggered. If *doNotUpdate* is null, undefined, or false, this method triggers a `modelChanged` event on the reference. An event object with an event of "updateAll" is passed to the `modelChanged` event.

**Note:** In Central 1.5 you can add a listener to an `LCDataProvider` server object using the `addListener()` method, but the listener object does not receive `modelChanged` events afterwards. If your agent needs event listening capability you can create a subclass of the `LCDataProvider` class and add your own event dispatching capabilities to the subclass.

## LCDataProvider.close()

### Availability

Macromedia Central Player 1.5.

### Edition

Macromedia Central SDK 1.5.

### Usage

```
close()
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; frees the communication resources associated with an LCDataProvider object. The `close()` method should be called in the `onDeactivate()` method of an application, agent, or pod, for the client and server sides of each LCDataProvider.

### Example

The following example shows how to use the `close()` method within an `onDeactivate()` method. The variable `myDP` represents an LCDataProvider client object.

```
function onDeactivate(Void):Void
{
    this.myDP.close();
}
```

## LCDataProvider.createClient()

### Availability

Macromedia Central.

### Usage

```
LCDataProviderInstance = mx.central.data.LCDataProvider.createClient(
    clientId, name);
```

### Parameters

*clientId* String; to uniquely identify this client. Use a combination of the second and third parameters received in your `onActivate()` callback.

*name* String; the same name passed when the server calls `createServer()`.

### Returns

An instance of the LCDataProvider object (client version).

## Description

`LCDataProvider` method; creates an instance of the data provider client. You should call this method from inside the `onActivate()` callback in your application and pods. You will want to save the reference returned from this method in a variable. You can use this variable to set the `dataProvider` property of your components. If you want to modify the source data, just modify the variable as if it was the original data. For example, to add an item, use `LCDataProviderInstance.addItem()`, or to sort, use `LCDataProviderInstance.sort()`. All subscribing components (in any application or pod) get updated automatically.

## LCDataProvider.createServer()

### Availability

Macromedia Central.

### Usage

```
LCDataProviderInstance = mx.central.data.LCDataProvider.createServer(name);
```

### Parameters

*name* String; the globally unique name of this data provider.

### Returns

An instance of the `LCDataProvider` object (server version).

## Description

`LCDataProvider` method; creates an instance of the data provider server. Call this method from inside the `onActivate()` callback method in your agent. The `createClient()` method calls from your application and pods need to provide the same name used in your `createServer()` call.

You will want to save the reference returned from this method in a variable. This variable is used to populate and modify the source data. The underlying data type of an `LCDataProvider` is a specialized version of an array. You can initialize the variable the same way as an array. However, when you want to add an item, don't use `push()`; use `addItem()` instead. Components expect each item in an `LCDataProvider` array to contain an object with properties for `label` and `data`. By creating even more complex data structures (basically, arrays that contain ActionScript objects with several properties), you can use the `DataProvider` to populate the `DataGrid` component or other components of your own design.

For more information about the `DataProvider` interface, see the Macromedia Flash documentation.



## LCDataProvider.getItems()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItems()
```

### Parameters

None.

### Returns

A reference to an array.

### Description

Method; returns the entire items array. This method returns only a reference to the actual *dataProviderInstance.items* array. Any changes made directly to the returned array will likely cause problems in the data provider. Use this accessor method for read-only purposes.

### Example

The following example traces the reference returned:

```
trace(getItems());
```

## LCDataProvider.getIndexByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getIndexByKey(key, value)
```

### Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

### Returns

An integer, or, if no items are found, -1.

### Description

Method; returns the index of the first item (starting from the beginning of the list) whose property indicated by *key* matches the *value* passed in.

## Example

The following example gets the index of the item added:

```
// adds two items to the list
someDp.addItem({category:"recipes", name:"salads"});
someDp.addItem({category:"recipes", name:"desserts"});

// returns the index of the first item encountered with the matching key/value
pair
var myIndex = someDp.getIndexByKey("category", "recipes");
trace(myIndex);
```

## LCDataProvider.getIndicesByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getIndicesByKey(key, value)
```

### Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

### Returns

An array of indexes or, if no items are found, an empty array.

### Description

Method; returns an array of indexes for items whose property specified by *key* matches *value*.

## Example

The following example gets the indexes of the items added:

```
// adds two items to the list
someDp.addItem({category:"movies", name:"thrillers"});
someDp.addItem({category:"movies", name:"comedies"});

// returns the indexes of the first item encountered with the matching key-
value pair
var myIndexes = someDp.getIndexByKey("category", "movies");
trace(myIndexes);
```

## LCDataProvider.getItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemAt(index)
```

### Parameters

*index* The index of the item to get.

### Returns

An object, or, if an invalid index is used, `undefined`.

### Description

Method; gets the item at *index* and returns the item.

### Example

The following code adds two items to the list and gets the second item:

```
myDp.addItem("label0", "data0");  
myDp.addItem("label1", "data1");  
var myItem:Object = myDp.getItemAt(1);
```

## LCDataProvider.getItemByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemByKey(key, value)
```

### Parameters

*key* A string, such as a property or label.

*value* A primitive data type value.

### Returns

An object or, if no item is found, `null`.

### Description

Method; returns the first item object found whose property specified in *key* matches *value*.

## Example

The following example adds two items and returns the first item:

```
myDp.addItem({category:"circus", name:"tiger"});
myDp.addItem({category:"circus", name:"lion"});
// returns the first object that matches the property key
// (in this case, the item named "tiger")
var myItem:Object = myDp.getItemByKey("category", "circus");
```

## LCDataProvider.getItemID()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemID(index)
```

### Parameters

*index* The index of the item whose index is retrieved.

### Returns

A string.

### Description

Method; returns the value of the `item._ID_` property found at *index*. The returned value is a string data type but can be evaluated to a number.

## Example

The following example gets the ID of the third item in the list:

```
var itemID:String = getItemID(2);
```

## LCDataProvider.getItemsByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getItemsByKey(key, value)
```

### Parameters

*key* A string.

*value* A primitive data type value.

## Returns

An array of objects, or, if no items are found, an empty array.

## Description

Method; returns an array of objects whose property specified in *key* matches *value*. The items returned are in the same order (relative to one another) as in the data provider.

## Example

The following example adds two items to the list and returns them both:

```
// returns both objects that match the property key
myDp.addItem({category:"hobbit", name:"Frodo"});
myDp.addItem({category:"hobbit", name:"Bilbo"});
var matchingItems:Array = myDp.getItemsByKey("category", "hobbit");
```

## LCDataProvider.getLength()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
getLength()
```

### Parameters

None.

### Returns

An integer.

### Description

Method; returns the total number of items in the data provider.

### Example

The following example traces the total number of items in the list:

```
var qty:Number = myDp.getLength();
trace("There are " + qty + " items in the list.");
```

## LCDataProvider.getSortState()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

**Usage**

`getSortState()`

**Parameters**

None.

**Returns**

A sort object with two string properties.

**Description**

Method; returns a sort object with two string properties, *sortField* and *order*:

- *sortField* is the property name last used for sorting or last passed to the `sort` method.
- *order* is the order property. Its value depends on what, if anything, was last passed to the `sort()` or `sortItemsBy()` methods. Possible values are "ASC", "DESC", the value undefined; or an empty string, "".

If the properties have no value, they will show as empty strings.

**Example**

The following example gets and displays the current sort state values, where `myDP` represents a `LCDataProvider` server object:

```
var sortObject:Object = this.myDP.getSortState();
trace("sortField=" + sortObject.sortField);
trace("order=" + sortObject.order);
```

## **LCDataProvider *listener.modelChanged()***

**Availability**

Macromedia Central Player.

**Edition**

Macromedia Central SDK.

**Usage**

```
listenerObject = new Object();
listenerObject.modelChanged = function(eventObject){
    // insert your code here
}
myDataProvider.addListener(listenerObject)
```

**Parameters**

*eventObject* An object with additional properties.

**Returns**

Nothing.

## Description

Event; broadcasts when the data provider changes. The `modelChanged()` method is not a method of the `LCDataProvider`; it is a function called *by* the `LCDataProvider` on an object registered as a view or listener object.

The *eventObject* object has the following properties:

- *source* A reference back to the data provider instance that initiated the event.
- *event* A string identifying the type of change made so that any listening views can perform granular updates. The following strings are possible values: "updateAll", "updateRows", "deleteRows", "sort", or "addRows".
- *firstRow* The number of the first row affected by the change.
- *lastRow* The number of the last row affected by the change.

**Note:** The *firstRow* and *lastRow* parameters are the same if only one row was affected. They are omitted if the event is "sort" or "updateAll".

## LCDataProvider.removeAll()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeAll()
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; removes all items in the data provider.

### Example

The following code removes all items:

```
myDp.removeAll();
```

## LCDataProvider.removeItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeItemAt(index)
```

### Parameters

*index* The index of the item to remove.

### Returns

An object.

### Description

Method; removes the item at *index*, and returns the removed item.

### Example

The following code removes the third item in the list:

```
var removedItem:Object = myDp.removeItemAt(2);
```

## LCDataProvider.removeListener()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
removeListener(listenerObj)
```

### Parameters

*listenerObj* A reference to an object.

### Returns

Nothing.

### Description

Method; searches the `_listeners` array for the reference passed in, and removes it if found.

### Example

The following code removes the listener from the LCDataProvider instance named `myDp`:

```
myDp.removeListener(myListener);
```



## LCDataProvider.replaceAllItems()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
replaceAllItems(items)
```

### Parameters

*items* An array of objects (ordered) or an object of objects (unordered).

### Returns

Nothing.

### Description

Method; deletes all of the items in the list and replaces them with the items specified in the *items* parameter.

## LCDataProvider.replaceItemAt()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
replaceItemAt(index, itemObj)
```

### Parameters

*index* The index of the item to be replaced.

*itemObj* An object. If the *itemObj* parameter is not typeof "object", then the argument is assigned to the item's label property instead of replacing the whole item (this behavior is slightly different than the LCDataProvider.updateItemByIndex() method).

### Returns

Nothing.

### Description

Method; overwrites an item at the specified *index* with a new item object. If the item passed in has an `_ID_` property, that property is overwritten with a new one.

## Example

The following example overwrites the object at index location 3 with an object containing values for the properties "category" and "name":

```
myDp.replaceItemAt(3, {category: "circus" , name: "lion"});
```

## LCDataProvider.setData()

### Availability

Macromedia Central.

### Usage

```
setData( uniqueArray );
```

### Parameters

*uniqueArray* Array; contains the source initial data for your data provider. Each item must have the same structure. Strings or numbers are acceptable. When your array contains ActionScript objects, each must have matching named properties.

### Returns

Nothing.

### Description

LCDataProvider method; use once to identify the structure for the LCDataProvider. It's easiest to use `setData()` to also populate the LCDataProvider with its initial data too, because that establishes the structure as well. That is, the structure matches whatever form the initial data has. The main thing is that you call `setData()` once to point to the *form* that the DataProvider is to take.

## Example

The following example uses the `setData()` method to associate the array with the LCDataProvider instance:

```
//for your agent:
onActivate=function(agentManager, agentID)
{
    //create an LCDataProvider instance to store in the myDP variable
    this.myDP = mx.central.data.LCDataProvider.createServer("my_data");
    //create and populate an array with identical Objects
    var myArray = [];
    myArray.push({first:"George", last:"Bush", gender:"MALE"});
    myArray.push({first:"Bill", last:"Clinton", gender:"MALE"});
    myArray.push({first:"Margaret", last:"Thatcher", gender:"FEMALE"});

    //associate
    this.myDP.setData( myArray );
};

//for your client
onActivate=function(shell, appID, shellID)
```

```

{
    //create a unique ID
    this.myID = appID+"_"+shellID;

    //create an LCDataProvider instance to store in the myDP variable
    myDP = mx.central.data.LCDataProvider.createClient(myID, "my_data");

    //link up an MDataGrid instance
    myDataGrid.dataProvider = this.myDP;

    //make a button to add items to DataProvider via input text fields
    add_btn.onPress=function()
    {
        var theItem={first: first_txt.text,
                      last: last_txt.text,
                      gender: gender_txt.text};
        myDP.addItem(theItem);
    };
};

```

## LCDataProvider.setItemByKey()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
setItemByKey(key, item)
```

### Parameters

*key* A string.

*item* An object.

### Returns

The index of the overwritten item.

### Description

Method; overwrites an existing object in the data provider with the *item* object provided. The object to be overwritten is chosen by finding the first item with a *key* property whose value matches the provided item object's *key* property value.

In addition, the item object that is provided must contain a property named `__ID__` whose value that matches the internally-assigned `__ID__` value of the item to be replaced.

**Note:** This method is primarily for internal use and might be deprecated in the future.

## Example

The following code adds an item and then overwrites the same item using the `setItemByKey()` method:

```
myDP.addItem({label:"bear", name:"Grizzly"});

var index:Number = myDP.setItemByKey("label", {label:"bear", name:"Teddy",
    __ID__:0});
```

## LCDataProvider.sort()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
sort(sortFunction[,key, order])
```

### Parameters

*sortFunction* A function.

*key* A string that should correspond to the name of an item property that you have defined.

*order* A string, "ASC" for ascending order or "DESC" for descending order.

### Returns

Nothing.

### Description

Method; sorts the items in a list using a custom function, similar to `Array.sort()`, but lets you also pass in the additional `key` and `order` parameters which will be stored by the `LCDataProvider` object and which can be retrieved later using the `getSortState()` method.

The `key` and `order` parameters are not passed to the specified `sortFunction`. The ordering of the items is performed in a second pass after the `sortFunction` has completed.

The `sort()` method can only be used on a `LCDataProvider` server object (which is usually created and managed in an agent). The `sort()` method will not work on an `LCDataProvider` client object.

**Note:** This method uses `ActionScript` to execute the sort, which is inherently slow. For large data sets, you might want to use the `sortItemsBy` method, which uses the native C++ implementation of the `Array.sortOn` method.

## Example

The following code sorts the list by the `name` property in ascending order, using a predefined sort function named `mySortFunction`:

```
myDp.sort(mySortFunction, "name", "ASC");
```

## LCDataProvider.sortItemsBy()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
sortItemsBy(key, order)
```

### Parameters

*key* A string that should map to an item property that you have defined.

*order* A string, "ASC" for ascending order, or "DESC" for descending order.

### Returns

Nothing.

### Description

Method; sorts the items in the specified order, using the built-in Array.sortOn method, with the item property specified in *key*.

This method works only for items that have a primitive data type in the specified key.

**Note:** This method uses the native C++ implementation of the Array.sortOn method. That implementation sorts items by their ASCII codes, which means that all lowercase characters appear before uppercase characters. For example, lowercase "z" is placed before uppercase "A". To sort in proper alphabetical order, you can create a new item property that has each item in uppercase letters, and sort on that property. This implementation will likely be faster than using a custom sort function.

### Example

The following example sorts items by their `name` property in ascending order:

```
myDp.sortItemsBy("name", "ASC");
```

## LCDataProvider.updateItem()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateItem(item)
```

### Parameters

*item* The existing object that overwrites the internal item.

### Returns

The index into which the item was passed.

### Description

Method; passes an existing item object to the data provider to overwrite an internal item with the same `_ID_` property.

## LCDataProvider.updateItemByIndex()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateItemByIndex(index, item)
```

### Parameters

*index* The index of the item to be overwritten.

*item* The new item object.

### Returns

The index of the overwritten item.

### Description

Method; overwrites the existing item object at *index* with a new item object. If the item object passed in does not have an `_ID_` property, an `_ID_` property is added to the new item object.

## LCDataProvider.updateView()

### Availability

Macromedia Central Player.

### Edition

Macromedia Central SDK.

### Usage

```
updateView(view, eventObject)
```

### Parameters

*view* A reference to an object.

*eventObject* A valid event object.

### Returns

Nothing.

## Description

Method; broadcasts the `modelChanged` event to all registered listeners on the current side of the `LCDataProvider` connection. For example, if the `updateView()` method is called on an `LCDataProvider` client object, the `modelChanged` event will be broadcast to every listener that was explicitly added to that client object.

The method called on each listener is:

```
view.modelChanged(eventObject:Object);
```

**Note:** This method does not broadcast the `modelChanged` event from a client object to a server object, or vice versa. Listeners on both sides of the `LCDataProvider` connection will normally receive `modelChanged` events when data is changed using the `DataProvider` API methods such as `addItem()` and `removeItem()`.

## LCService object

**ActionScript Class Name** mx.central.data.LCService

The `LCService` object provides a multiplexing service, which means that it can manage multiple requests called from and responses sent to the elements in your application (agent, application, and pods). The `LCService` object insulates you from the work involved to overcome several particularly difficult limitations in the traditional Local Connection object. Specifically, `LCService` requests can be made synchronously, which means that the response gets returned immediately (instead of needing to first define a callback, as is the case with the asynchronous behavior in regular Local Connection objects). Additionally, the `LCService` object alleviates you from managing a unique channel name for each element's sends and responses. For example, an agent, application, and pod normally need six unique channel names to pass messages among the pieces. The `LCService` object greatly simplifies this by letting you identify a single channel for all pieces.

In practice, this means that you can store code that's shared among your application elements in one place. The agent can manage the global methods, and each application or pod instance can have its own implementation of their own methods. A client/server metaphor is used (like the [LCDataProvider object](#)). Your agent is the server, and each application and pod is a client. The basic approach is to identify the methods that will reside in the agent (that is, server) and the methods residing in the application and pods (the clients). Clients can call methods in the server and the server can trigger methods in the clients. These methods can also gracefully pass data back. In the end, one element in your application can trigger code that resides in another element nearly as easy as it can trigger code in its own file. In the most simple sense, this means that your code doesn't need to be duplicated, which makes it easier to manage and troubleshoot.

Because you can have several instances of clients (application and pods), the code you designate as residing in the client *will* be duplicated. Normally, you just store common code in one place—the server. However, defining methods that reside in the client gives you the opportunity to have each piece *implement* the same named method in its own manner. For example, when the server says "doTheMethod()", your application could do one thing and the pod something else.

There are three pieces to LCService: the server, the client, and the interface. The interface is simply the list of methods you plan to use for the server and the list for the client. The interface defines how the client and server interface with each other. Because clients and servers need an identical interface, you should create a script file, as in the following procedure, and use `#include` so that each element (agent, application, pod) can point to the same source.

#### To use LCService:

1. Create an interface object by putting the following code into a text file named `interface.as`, so that each file can share it (using `#include`). Store in a variable an `Object` with the properties `name` (containing a string) and `interfaces` (containing *another* object with the properties `Client` and `Server`, which both contain an array of strings matching your method names):

```
myInterface = new Object();
myInterface.name = "myInterfaceName";
myInterface.interfaces = new Object();

myInterface.interfaces.Client = ["clientMethod1"];
myInterface.interfaces.Server = ["serverMethod1", "serverMethod2"];
```

2. In the server (that is, the agent SWF file), define methods that match the names used in the preceding step, as you would normally, as follows:

```
this.serverMethod1=function( param )
{
    return "server heard you say "+param;
};
this.serverMethod2=function()
{
    //perform some task
};
```

3. In the clients (that is, the application and pod files), define methods that match the names used previously, as you would normally. The following code shows two different implementations—one for the application and one for the pod:

```
//in the application
this.clientMethod1=function( param )
{
    gShell.setStatus(param);
    message_txt.text="Server said "+param;
};
//in the pod
this.clientMethod1=function( param )
{
    message_txt.text="Pod heard server say "+param;
};
```

4. Define the server side in your agent by calling `mx.central.data.LCService.createServer()`, as follows:

```
//make sure the myInterface variable gets set first
#include "interface.as"

onActivate=function()
```



```

{
    //first parameter is the variable containing the interface object
    //second parameter is the object where the method definitions can be found
    //third parameter specifies we want the LCService to be synchronous
    myService=mx.central.data.LCService.createServer(myInterface, this, true
    );
};

```

5. Define the client side in your application and pods by calling `LCService.createClient()`, as follows:

```

//make sure the myInterface variable gets set first
#include "interface.as"

onActivate=function(shell, appID, shellID)
{
    //create a unique ID
    myID = appID+"_"+shellID;

    //first parameter is the variable containing the interface object
    //second parameter is a unique string for this application/pod instance
    //third parameter is the object where the method definitions can be found
    //third parameter specifies we want the LCService to be synchronous

    myService=mx.central.data.LCService.createClient(myInterface, myID, this,
    true);
};

```

6. The client can call server methods by preceding the function names with a reference to the `LCService` instance created, as follows:

```

//trigger one with a return value expected
message_txt.text=myService.serverMethod1("some value");

//call one with no return value expected
myService.serverMethod2();

```

7. Similarly, the server can invoke the methods in each connected client in a similar manner, as follows:

**Note:** The variable name `myService` happens to match.

```

//trigger a method in all the connected clients and pass a string
myService.clientMethod1("server is talking");

```

## Method summary for the `LCService` object

Method	Description
<code>LCService.createClient()</code>	Used in your application or pod file to create an instance of the <code>LCService</code> client. This way you can trigger methods in the server and the server can trigger your methods.
<code>LCService.createServer()</code>	Used in your agent file to create an instance of the <code>LCService</code> server. This way the clients can trigger methods in the agent and the agent can trigger methods implemented in the clients.

## Property summary for the LCService object

Property	Description
None.	

## Event handler summary for the LCService object

Event handler	Description
None.	

## Interfaces

LCService uses Interface objects to declare which methods are implemented by which side of the connection. The extra step of defining an interface adds a layer of security; clients only have access to the Server methods published in the interface, and vice versa.

Interface is an ActionScript object with the following two properties:

Field	Description
<code>name</code>	String; globally unique string identifier of the Interface. This property is used by the clients and service to determine to which object they should be mutually connected.
<code>interfaces</code>	Object; contains two properties ( <code>Client</code> and <code>Server</code> ). These each contain an array of strings that match the methods that reside in the client and server elements (application or pod and agent, respectively).

It is crucial that the Server and all Clients use equivalent interface objects for LCService to work correctly. To avoid mismatched Interface object names, place your interfaces in an external ActionScript file to be included by every SWF file in your application.

## Synchronous versus Asynchronous

By default, the LCService object uses normal asynchronous LocalConnection objects. This means that in order to get a result from your function invocations, you need to define a result handler, which is called when your request completes. If you're familiar with traditional LocalConnection objects, you know that this approach involves more work. If you don't specify an `onResult()` callback, Central looks for a function that matches the pattern *funcName\_Result*.

Because Central always runs locally, you usually specify the *sync* parameter as `true` which means that you want to use the synchronous behavior. (The *sync* parameter is the last parameter in both the `createClient()` and `createServer()` methods.) However, if the server process you are calling takes a while to complete (for instance, it is making several web service calls) you might want an asynchronous connection. That way your application or pod can do other processes while the server does its work.

## Security

The default security behavior inherits from the LocalConnection object's security: only Clients and Servers from the same domain can communicate.

By defining an `allowDomain()` function on your Client and/or Server objects, as the following example shows, you can allow access from other domains:

```
gTestService = mx.central.data.LCService.createServer( myInterface, this );
allowDomain=function( domain )
{
    if( domain == this.domain() || domain == "frienddomain.com" )
    {
        return true;
    } else {
        return false;
    }
};
```

## LCService.createClient()

### Availability

Macromedia Central.

### Usage

```
myService = mx.central.data.LCService.createClient( interfaceObj, id,
    callbackObj, bSync );
```

### Parameters

*interfaceObj* Object; contains details about the LCService name and the list of methods implemented by both Client and Server. This should contain the same `name` property in both the client (here) and the server. For details about the format of an interface object, see [“Interfaces” on page 298](#).

*id* String; uniquely identifies this client. Use a combination of the second and third parameters received in your `onActivate()` callback.

*callbackObj* Object; specifies where the methods exposed from this client reside. Simply use this if you want to define your methods in the root of your application or pod SWF instance.

*bSync* A Boolean value; indicates whether the remote calls should be performed synchronously. Setting this to `true` makes many tasks much easier.

### Returns

A reference to the server that your application or pod can use to expose functions and trigger remote functions.

### Description

LCService method; creates a client-side local connection service, so that you can both trigger remote methods and expose methods in your SWF file for the server to call.

For general information about LCService, see [“LCService object” on page 295](#).

## LCService.createServer()

### Availability

Macromedia Central.

### Usage

```
myService = mx.central.data.LCService.createServer( interfaceObj, callbackObj,  
    bSync );
```

### Parameters

*interfaceObj* Object; contains details about the LCService name and the list of methods implemented by both Client and Server. This should contain the same `name` property in both the client (here) and the server. For details about the format of an interface object, see [“Interfaces” on page 298](#).

*callbackObj* Object; specifies where the methods exposed from this client reside. Simply use this if you want to define your methods in the root of your application or pod SWF instance.

*bSync* A Boolean value; indicates whether the remote calls should be performed synchronously. Setting this to `true` makes many tasks much easier.

### Returns

A reference to the server that your agent can use to expose functions and trigger remote functions.

### Description

LCService method; creates a server-side local connection service so that you can both trigger methods in the clients and expose methods in your SWF file so the clients can call them.

For general information about LCService, see [“LCService object” on page 295](#).

## Log object

**ActionScript Class Name** `mx.central.services.Log`

The Log object is part of the Central WebService API and is intended to be used with the WebService object (see [“WebService object” on page 384](#)).

You can create a new Log object to record activity related to a WebService object. To execute code that runs as messages are sent to a Log object, use the `onLog()` callback function. There is no log file; the logging mechanism has to be created by you with the code you put inside the `onLog()` callback, such as sending the log messages to a `trace()` command.

The constructor for this object returns a Log object that can then be passed as an optional argument to the WebService constructor (see [“WebService object” on page 384](#)).

**Flash MX 2004 ActionScript Class Name** `mx.central.services.Log`

## Method summary for the Log object

Method	Description
None.	

## Property summary for the Log object

Property	Description
None.	

## Event handler summary for the Log object

Event handler	Description
<a href="#">Log.onLog()</a>	Sends a log message to a log object.

## Constructor for the Log object

### Availability

Macromedia Central

### Usage

```
myWebSvcLog = new Log(logLevel [, logName]);
```

### Parameters

*logLevel* Log level to indicate the types of information you want to pass to the `onLog()` event. In the web services code, the log messages are broken down into categories or levels. The `logLevel` parameter of the Log object constructor relates to these categories. The following three `logLevels` are available:

- `Log.BRIEF`: The log records primary life-cycle event and error notifications.
- `Log.VERBOSE`: The log records all life-cycle event and error notifications.
- `Log.DEBUG`: The log records metrics and fine-grained events and errors.

The default `logLevel` is `log.BRIEF`.

*logName* Optional name that is included with each log message. If you are using multiple log objects, you can use the `logName` parameter to determine which log recorded a given message.

### Returns

Nothing.

### Description

Constructor; creates a Log object. Use this constructor to create a log. After you create the Log object, you can pass this object to a web service to get messages.

### Example

You can call on the new Log constructor, which returns a log object to pass to your web service:

```
// creates a new log object
myWebSvcLog = new Log();
myWebSvcLog.onLog = function(txt)
{
    trace(txt)
};
```

You then pass this Log object as a parameter to the WebService constructor:

```
myWebSvc = new WebService("http://www.mysite.com/info.wsd1", myWebSvcLog);
```

As the web services code executes, and messages are sent to the log object, the `onLog()` function of your Log object is called. This is the only place to put code that displays the log messages if you want to see them in real time.

The following are examples of log messages:

```
7/30 15:22:43 [INFO] SOAP: Decoding PendingCall response
7/30 15:22:43 [DEBUG] SOAP: Decoding SOAP response envelope
7/30 15:22:43 [DEBUG] SOAP: Decoding SOAP response body
7/30 15:22:44 [INFO] SOAP: Decoded SOAP response into result [16 millis]
7/30 15:22:46 [INFO] SOAP: Received SOAP response from network [6469 millis]
7/30 15:22:46 [INFO] SOAP: Parsed SOAP response XML [15 millis]
7/30 15:22:46 [INFO] SOAP: Decoding PendingCall response
7/30 15:22:46 [DEBUG] SOAP: Decoding SOAP response envelope
7/30 15:22:46 [DEBUG] SOAP: Decoding SOAP response body
7/30 15:22:46 [INFO] SOAP: Decoded SOAP response into result [16 millis]
```

## Log.onLog()

### Availability

Macromedia Central

### Usage

```
myWebSvcLog.onLog = function(message)
{
    //process the message
};
```

### Parameters

*message* String; the log message passed to the handler. For more information about log messages, see [“Log object” on page 300](#).

### Returns

Nothing.

### Description

Log callback function; Macromedia Flash Player calls this function when log messages are received. That is, you write this handler so that you can trigger code that processes the messages. This handler is a good place to put code that records or displays the log messages, such as a `trace()` command. The Log construction is described in [“Log object” on page 300](#).

### Example

The following example creates a new log object, passes it to a new WebService object and handles the logging messages:

```
// create a new log object
myWebSvcLog = new Log();
```

```
// pass the log object to the web service
myWebService = new WebService(myWSDLUri, myWebSvcLog);

// handle incoming log messages
myWebSvcLog.onLog = function(message)
{
    trace("Log Event:\r myWebSvcLog.message="+message+);
}
}
```

## MD5 object

**ActionScript Class Name** mx.central.encryption.MD5

An algorithm for encoding a string in a way that never exposes the content. It's called a *one way hash* because you can only encode. In practice, you encode a string and save the encoded version (called a digest). Later you can compare the digest to an encoded version of a new string you're trying to verify, to see if they match. This way you only save encoded strings that are theoretically impossible to decode.

### Method summary for the MD5 object

Method	Description
<a href="#">MD5.encode()</a>	Returns an MD5 Digest of a given string.

### Property summary for the MD5 object

Property	Description
None.	

### Event handler summary for the MD5 object

Event handler	Description
None.	

## MD5.encode()

### Availability

Macromedia Central.

### Usage

```
myDigest = MD5.encode( messageString );
```

### Example

```
//this example saves an encoded version of the password locally

//stop the user on first frame for verification
stop();
```

```

onActivate()
{
    mySO=SharedObject.getLocal("preferenceFile");

    //if they don't have a password make them create one, otherwise verify
    //(naturally, if they just remove the shared object file they bypass this)

    if(mySO.data.password==undefined)
    {
        okay_btn.label = "create password";
        okay_btn.onRelease = doNewPassword;
        input_txt.text="enter new password here";
    }
    else
    {
        okay_btn.label =("verify");
        okay_btn.onRelease = doVerify;
        input_txt.text="type saved password here";
    }
};

doNewPassword=function()
{
    //save an encoded version of what they entered in the shared object
    mySO.data.password=MD5.encode(input_txt.text);
    gotoAndStop("home_frame");
};

doVerify=function()
{
    //get the old password (already encoded);
    var oldEncodedPassword=mySO.data.password;

    //see if old encoded password matches an encoded version of what they type
    if(oldEncodedPassword==MD5.encode(input_txt.text))
    {
        gotoAndStop("home_frame");
    }
    else
    {
        input_txt.text="no, try again";
    }
};

```

## Parameters

*messageString*   String to encode.

## Returns

*digest*   String; the encoded version of the input string.



## Description

MD5 method; takes a string and returns a 128-bit “fingerprint” or “message digest” version. This method is useful for message integrity checking. You can use the `MD5.encode()` method from anywhere in Central. This is a one-way hash function. There is no way to take an MD5 hash and decrypt it or determine what string was used to generate it. So, it isn’t a way to store encrypted data, but rather a way to determine if data has changed.

Although the example shows a way of saving encoded data locally, MD5 is also good for limiting the chance that users’ sensitive data is compromised while under your control. It’s a way to ensure that data is never sent to you in a form that can be decoded. For example, you can store encoded versions of all users’ passwords in your database.

## MovieClip object

**ActionScript Class Name**    `MovieClip`

A special feature was added to Central that can let you identify a string that appears as a true (operating system level) tooltip when a user’s mouse hovers over a particular movie clip.

### Property summary for the MovieClip object

Property	Description
<code>MovieClip.tooltipText</code>	String that appears as a tooltip approximately one-half second after the user places their pointer over this clip.

### MovieClip.tooltipText

#### Availability

Macromedia Central.

#### Usage

```
myClipInstance.tooltipText = tipString;
```

#### Example

```
print_pb.tooltipText = "Print Records";
```

#### Parameters

*tipString*    The string that you want to appear as a tooltip.

#### Description

MovieClip property; only available in Central. The tip appears when the user holds the mouse over a movie clip, and goes away when the user clicks or moves the mouse off the movie clip. To turn off the behavior for a clip that already has a `tooltipText` property, set the value to an empty string, as the following code shows:

```
myClipInstance.tooltipText="";
```

This straightforward feature has just a few caveats. First, you cannot control exactly when the tooltip appears. Typically, you can expect a one-half second delay, but this can vary. Also, the entire bounding area of a clip is used. That is, an irregularly shaped clip will have an active “hit” area that’s a rectangle matching its maximum width and height (like the box you see when you select a clip while authoring). Also, even if the movie clip is currently obscured by another visual element, the tooltip is still active. In such cases, turn off the tooltip by setting it to an empty string. Finally, the maximum length of the tooltip’s string is based on the user’s operating system. However, best practices dictate that tooltips are not exceedingly verbose.

## PendingCall object

**ActionScript Class Name** `mx.central.services.PendingCall`

The PendingCall object is part of the `mx.central.services` package and is intended to be used with the WebService object (see [“WebService object” on page 384](#)).

When you call a method on a WebService object, the WebService object returns a PendingCall object. You don’t construct a PendingCall object; rather you just save the instance in a variable. Then you write `onResult` and `onFault` callbacks for that instance to handle the asynchronous response from the web service method. If the web service method returns a fault, Central calls the `pendingCallInstance.onFault` callback and passes a SOAPFault object that represents the XML SOAP fault returned by the server/web service. If the web service invocation is successful, Central calls the `pendingCallInstance.onResult` callback and passes a result object. The result object that arrives as the parameter for your `onResult` callback is the XML response from the web service decoded or deserialized into an ActionScript object. For more information about the WebService object, see [“WebService object” on page 384](#).

Additionally, the PendingCall object offers you access to output parameters when there are more than one. Many web services return only a single result, but some web services return more than one result. The return value referred to in this API is simply the first (or only) result. The `PendingCall.getOuptutXXX` functions give you access to all of the results, not just the first. Although the return value is handed to you as the only argument in the `onResult()` callback, if there are other output parameters that you need to access, use `getOutputValues()` (returns an Array) and `getOutputValue(index)` (returns an individual one) to get all the values (decoded into ActionScript objects).

Finally, you can also access the SOAPParameter object directly. The SOAPParameter object is an ActionScript object with two properties: `value` contains the ActionScript value of an output parameter; `element` contains the XML value of the output parameter. The following functions return a SOAPParameter object, or an array of SOAPParameter objects, which contains the value (`param.value`) as well as the XML element (`param.element`): `getOutputParameters()`, `getOutputParameterByName(theName)`, and `getOutputParameter(theIndex)`.

**Flash MX 2004 ActionScript Class Name** `mx.central.services.PendingCall`

## Method summary for the PendingCall object

Method	Description
<code>PendingCall.getOutputParameter()</code>	Gets a SOAPParameter object based on the <i>index</i> you provide.
<code>PendingCall.getOutputParameterByName()</code>	Gets a SOAPParameter object based on the <i>localName</i> passed in.
<code>PendingCall.getOutputParameters()</code>	Gets an array of SOAPParameter objects.
<code>PendingCall.getOutputValue()</code>	Gets the output value based on the <i>index</i> passed in.
<code>PendingCall.getOutputValues()</code>	Gets an array of all the output values.

## Property summary for the PendingCall object

Property	Description
<code>PendingCall.myCall</code>	The SOAPCall operation descriptor for the PendingCall operation.
<code>PendingCall.request</code>	The SOAP request in raw XML format.
<code>PendingCall.response</code>	The SOAP response in raw XML format.

## Event handler summary for the PendingCall object

Event handler	Description
<code>PendingCall.onFault()</code>	Called by a web service when the method fails.
<code>PendingCall.onResult()</code>	Called when a method has succeeded and returned a result.

## Constructor for the PendingCall object

### Availability

Macromedia Central

### Description

The PendingCall object is not constructed by the developer. Instead, when you call a remote function on a WebService object, the WebService object returns a PendingCall object. That is, replace PendingCall with the instance name of your PendingCall instance (returned when constructing a new WebService).

## PendingCall.getOutputParameter()

### Availability

Macromedia Central

### Usage

```
mySOAPParameterObj=myPendingCall.getOutputParameter(index);
```

## Parameters

*index* The index of the parameter.

## Returns

SOAPParameter object with the following elements:

Element	Description
value	An ActionScript object that contains the value of the parameter.
element	The raw XML of the parameter in the SOAP envelope.

## Description

Function; gets an additional output parameter of the SOAPParameter object, which contains the value and the XML element. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always handed to you as the single results argument of the `onResult()` callback, but to get access to the others you need to use functions such as this one or `getOutputValue()`. The `getOutputParameter()` function returns the *n*th output parameter as a SOAPParameter object.

See also [PendingCall.getOutputValue\(\)](#), [PendingCall.getOutputValues\(\)](#), [PendingCall.getOutputParameterByName\(\)](#), and [PendingCall.getOutputParameters\(\)](#).

## Example

Given the following SOAP descriptor file, `getOutputParameter(1)` would return a SOAPParameter object with `value="Hi there!"` and `element=the <outParam2> XMLNode:`

```
...
<SOAP:Body>
  <rpcResponse>
    <outParam1 xsi:type="xsd:int">54</outParam1>
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>
    <outParam3 xsi:type="xsd:boolean">true</outParam3>
  </rpcResponse>
</SOAP:Body>
...
```

## PendingCall.getOutputParameterByName()

### Availability

Macromedia Central

### Usage

```
mySOAPParameterObj=myPendingCall.getOutputParameterByName(localName);
```

### Parameters

*localName* The local name of the parameter. In other words, the name of an XML element, stripped of any namespace information. For example, the local name of both of the following elements is *bob*:

```
<bob abc="123">  
<xsd:bob def="ghi">
```

## Returns

SOAPParameter object with the following elements:

Element	Description
value	An ActionScript object that contains the value of the parameter.
element	The raw XML of the parameter in the SOAP envelope.

## Description

Function; gets any output parameter as a SOAPParameter object, which contains the value and the XML element. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one. The `getOutputParameterByName()` call returns the output parameter with the name *localName*.

See also [PendingCall.getOutputValue\(\)](#), [PendingCall.getOutputValues\(\)](#), [PendingCall.getOutputParameter\(\)](#), and [PendingCall.getOutputParameters\(\)](#).

## Example

Given the following SOAP descriptor file, `getOutputParameterByName("outParam2")` would return a SOAPParameter object with a value equal to "Hi there!" and an element in the form of an XMLNode equal to `<outParam2>`:

```
...  
<SOAP:Body>  
  <rpcResponse>  
    <outParam1 xsi:type="xsd:int">54</outParam1>  
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>  
    <outParam3 xsi:type="xsd:boolean">true</outParam3>  
  </rpcResponse>  
</SOAP:Body>  
...
```

## PendingCall.getOutputParameters()

### Availability

Macromedia Central

### Usage

```
arrayOfSOAPParameterObjects=myPendingCall.getOutputParameters()
```

### Parameters

None.

## Returns

Array of SOAPParameter objects with the following elements:

Element	Description
value	The value of the parameter in the form of an ActionScript object.
element	The raw XML of the parameter in the SOAP envelope.

## Description

Function; gets additional output parameters of the SOAPParameter object, which contains the values and the XML elements. SOAP RPC calls may return multiple output parameters. The first (or only) return value always arrives as the single argument for the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputValues()`.

See also [PendingCall.getOutputValue\(\)](#), [PendingCall.getOutputValues\(\)](#), [PendingCall.getOutputParameterByName\(\)](#), and [PendingCall.getOutputParameter\(\)](#).

## PendingCall.getOutputValue()

### Availability

Macromedia Central

### Usage

```
oneOutputParameter=myPendingCall.getOutputValue(index);
```

### Parameters

*index* The index of an output parameter. The first parameter is index 0.

### Returns

The *nth* output parameter matching the *index* that you specify.

### Description

Function; gets the decoded ActionScript value of an individual output parameter. SOAP RPC calls may return multiple output parameters. The first (or only) return value is always handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputParameter()`. The `getOutputValue()` call returns the *nth* output parameter.

See also [PendingCall.getOutputParameter\(\)](#), [PendingCall.getOutputValues\(\)](#), [PendingCall.getOutputParameterByName\(\)](#), and [PendingCall.getOutputParameters\(\)](#).

### Example

Given the following SOAP descriptor file, `getOutputValue(2)` would return true:

```
...
<SOAP:Body>
  <rpcResponse>
    <outParam1 xsi:type="xsd:int">54</outParam1>
    <outParam2 xsi:type="xsd:string">Hi there!</outParam2>
```

```

        <outParam3 xsi:type="xsd:boolean">true</outParam3>
    </rpcResponse>
</SOAP:Body>
...

```

## PendingCall.getOutputValues()

### Availability

Macromedia Central

### Usage

```
arrayOfOutputParams=myPendingCall.getOutputValues();
```

### Parameters

None.

### Returns

Array of all output parameters' decoded values.

### Description

Function; gets the decoded ActionScript value of all output parameters. SOAP RPC calls can return multiple output parameters. The first (or only) return value is always handed to you in the results argument of the `onResult()` callback, but to get access to the others you need to use APIs such as this one or `getOutputParameters()`.

See also [PendingCall.getOutputValue\(\)](#), [PendingCall.getOutputParameter\(\)](#), [PendingCall.getOutputParameterByName\(\)](#), and [PendingCall.getOutputParameters\(\)](#).

## PendingCall.myCall

### Availability

Macromedia Central

### Usage

```
myPendingCall.myCall
```

### Description

Property; the SOAPCall object corresponding to the PendingCall object's operation. The SOAPCall object contains information about the web service operation, and provides control over certain behaviors. The `myCall` property is the literal property name to use, not a holder for some name you provide. For more information, see ["SOAPCall object" on page 381](#).

### Example

The following `onResult` callback traces the name of the SOAPCall operation:

```

myCallback.onResult = function(result)
{
    // Check my operation name
    trace("My operation name is " + this.myCall.name);
};

```

## PendingCall.onFault()

### Availability

Macromedia Central

### Usage

```
myPendingCallObj.onFault = function(fault)
{
    // handles any faults, for example, by telling the
    // user that the server isn't available or to contact technical
    // support
};
```

### Parameters

*fault* Decoded ActionScript object version of the error containing the properties in the following list. If the error information came from a server in the form of XML, the SOAPFault object is the decoded ActionScript version of that XML.

The type of error object returned to `PendingCall.onFault()` is a SOAPFault object. It is not constructed by the Central developer, but returned as the result of a failure. This object is an ActionScript mapping of the SOAP Fault XML type.

SOAPFault property	Description
<code>faultcode</code>	String; the short standard string describing the error.
<code>faultstring</code>	String; the human-readable description of the error.
<code>detail</code>	String; the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
<code>element</code>	XML; the XML object representing the XML version of the fault.
<code>faultactor</code>	String; the source of the fault (optional if an intermediary is not involved).

### Returns

Nothing.

### Description

PendingCall object callback function; you provide this handler that Flash Player calls when a web service method has failed and returned an error. The *fault* parameter is an ActionScript SOAPFault object.

### Example

The following example handles errors returned from the web service method:

```
// handles any error returned from the use of a web service method
myPendingCallObj = myWebService.methodName(params)
myPendingCallObj.onFault = function(fault)
{
    // catches the SOAP fault
    debugOutputField_txt.text = fault.faultstring;
    // add code to handle any faults, for example, by telling the
```



```
        // user that the server isn't available or to contact technical
        // support
    };
```

## PendingCall.onResult()

### Availability

Macromedia Central

### Usage

```
myPendingCallObj.onResult = function(result)
{
    // catches the result and handles it for this application
}
```

### Parameters

*result* An ActionScript object version of the XML result returned by a web service method called with `myPendingCallObj = myWebService.methodName(params)`.

### Returns

Nothing.

### Description

PendingCall callback function; you provide this handler in order to trap the results returned from the remote methods you call. The result is a decoded ActionScript object version of the XML returned by the operation. To get the raw XML returned instead of the decoded result, access the `PendingCall.response` property (see [“PendingCall.response” on page 314](#)).

### Example

The following example handles results returned from the web service method:

```
// handles results returned from the use of a web service method
myPendingCallObj = myWebService.methodName(params)
myPendingCallObj.onResult = function(result)
{
    // catch the result and handle it for this application
    ResultOutputField.text = result;
}
```

## PendingCall.request

### Availability

Macromedia Central

### Usage

```
rawXML = myPendingCallback.request;
```

## Description

PendingCall property; contains the raw XML form of the current request sent with `myPendingCallback = myWebService.methodName()`. Typically, `PendingCall.request` provides more information than needed, but you can use it if you are interested in the SOAP that gets sent over the wire. Use this property to access the raw XML of the request. Use `myPendingCallback.onResult()` to get the ActionScript version of the results of the request.

## PendingCall.response

### Availability

Macromedia Central

### Usage

```
rawXML = myPendingCallback.response;
```

### Description

PendingCall property; contains the raw XML form of the response to the most recent web service method call sent with `myPendingCallback = myWebService.methodName()`. Normally, you would not have any use for `PendingCall.response`, but you can use it if you are interested in the SOAP that gets sent over the wire. Because the initial call is asynchronous and it will take some time, be sure not to attempt to access the response property until the `onResult()` callback is triggered. Use `myPendingCallback.onResult()` to get the corresponding ActionScript version of the results of the request.

## Pod object

**ActionScript Class Name** `mx.central.Pod`

The Pod object is equivalent to your pod SWF instance. That is, your pod SWF instance becomes an instance of the Pod object. The methods listed next give your pod a way to return information to Central when requested. The event handlers listed give your pod a way to react to global events triggered by Central. To use any of these, simply replace *Pod* with *this* (provided you're in your pod SWF instance).

## Method summary for the Pod object

Method	Description
<code>Pod.getLastTabIndex()</code>	Called by the Console to ascertain the highest tab index that your pod is using (so that other pods know where to start tabbing). This is a function you write to return information to Central in order to maintain a good user experience.
<code>Pod.setBaseTabIndex()</code>	Called by the Console whenever the <code>baseTabIndex</code> of your pod has changed so that you can update the <code>tabIndex</code> for controls such as text and buttons.

## Property summary for the Pod object

Property	Description
None.	

## Event handler summary for the Pod object

Event handler	Description
<code>Pod.onActivate()</code>	Called by the Console when your pod has instantiated.
<code>Pod.onDeactivate()</code>	Called by the Console when your pod is about to unload.
<code>Pod.onNetworkChange()</code>	Called by the Console when the connection status changes.
<code>Pod.onNoticeEvent()</code>	Called by the Console when a notice created by your application is engaged or closed by the user or gets removed programmatically using a script or a time-out.
<code>Pod.onPositionChange()</code>	Called by the Console when the user drags a pod to a new position, when the user adds a pod to the Console, or when the user removes another pod above the current pod.
<code>Pod.onSelectedItem()</code>	Called by the Console when your pod receives Blast data. In the case of pods, data can only be received when the user selects All On Screen from the Blast pop-up menu. The application sending data always needs to first make a selection of a data type that matches one listed in your products.xml file's <code>supportedTypes</code> tag. Note: Both applications and pods, or pod classes, need a <code>supportedTypes</code> tag in order to receive Blast data.

## Pod.getLastTabIndex()

### Availability

Macromedia Central.

### Usage

```
getLastTabIndex=function()  
{  
    return gMaxTabIndexUsedByMyPod;  
}
```

### Parameters

None.

### Returns

An integer representing the last tab index used for this pod.

## Description

Pod callback method; called by the Console to determine the highest tab index that your pod is using. You should write this handler so that your pod can report back to the Console. This way, other pods can be given the next logical tab index and the user can easily tab through the entire Central environment. You do this as a courtesy, but it also ensures a good user experience.

## Pod.onActivate()

### Availability

Macromedia Central.

### Usage

```
onActivate = function(console, podID, viewerID, position, baseTabIndex,
    initialData)
{
    //set a variable to reference the Console
    gConsole = console;

    //trigger our own onNetworkChange handler
    //using the current connection status
    this.onNetworkChange(gConsole.isConnected());

    //create a unique name for use with LocalConnection
    gUniqueName=podID+"_"+viewerID;

    //set the starting tabIndex so our app can be accessible
    gBaseTab=baseTabIndex;
    username_txt.tabIndex=gBaseTab;
    password_txt.tabIndex=gBaseTab+1;
    continue_btn.tabIndex=gBaseTab+2;

    //display an optional message based on the pod that was created
    if(initialData!=undefined)
    {
        message_txt.text="Thanks for making a "+initialData+" pod";
    }
};
```

### Parameters

*console* Console object; use this to reference any functions in the Central Console API.

*podID* Number; a unique ID for this particular pod installed with your application. Although the same pod can appear multiple times, each instance will have the same *podID*.

*viewerID* Number; a unique ID for this pod instance. Combined with the *podID* parameter, you can create a unique string identifier to use with the LocalConnection object.

*position* Integer; specifies the ordinal position of the pod (not the pixel location). All pods have the *position* 0 when they first appear, because they always show up in the top title *position*. However, because `onActivate()` also triggers when the pod is reopened, *position* won't always be 0.

*baseTabIndex* Number; used for accessibility—your pod should set the tab indexes on controls such as buttons and text fields starting with this number, so it doesn't interfere with surrounding Console controls (such as the toolbar).

*initialData* Any data type; passed to your pod at launch (either in the product.xml file's *initialData* tag or sent as a parameter in the `addPod()` method).

## Returns

Nothing.

## Description

Console callback event; called by the Console when a pod is instantiated. When your pod SWF instance calls `Central.initPod()`, this method is called when initialization is complete.

You pass *initialData* by declaring it in your application's product.xml file using the *initialData* tag. Alternatively, an application or other pod can send *initialData* when it calls `addPod()`. This way the data can be dynamic.

A best practice is to keep a reference to the console (*gConsole* in the following example), so that you have an object onto which you can attach subsequent calls to the Console API:

```
gConsole = console;
if(!gConsole.inLocalInternetCache("http://www.mysite.com/my_photo.jpg")){
    gConsole.addToLocalInternetCache("http://www.mysite.com/my_photo.jpg");
}
```

In addition, it's a good practice to trigger all the handlers that keep your application refreshed, once in the `onActivate()` handler. For example:

```
this.onNetworkChange(gConsole.isConnected());
```

**Note:** In order to consolidate the code samples shown for features common to the AgentManager, Shell, and Console, many examples show just the first parameter (*console* in this case) being saved in a variable (shown as *gConsole*). Although each of these three objects has a slightly different implementation of `onActivate`, they all start with a reference to the respective managing object.

## Pod.onDeactivate()

### Availability

Macromedia Central.

### Usage

```
onDeactivate=function()
{
    // perform clean up
    clearInterval(gMyInterval);
    mySharedObject.data.closingTime=new Date();
};
```

### Parameters

None.

## Returns

Nothing.

## Description

Agent, application, or pod event handler; called by the respective shell the instant before an agent, application, or pod instance is unloaded. Central triggers the `onDeactivate()` method each time the user uninstalls or updates an application, or exits Central. The `onDeactivate()` event should clean up any global references, including the following:

- Global variables
- Open network connections
- Open Local Connections
- Open `LCService` and `LCDataProvider` objects
- Events triggered by `setInterval` (using `clearInterval()`)

Code you place inside the `onDeactivate()` method is ensured to run and is also the last code to execute code before Central shuts down.

## Pod.onNetworkChange()

### Availability

Macromedia Central.

### Usage

```
onNetworkChange = function (connected)
{
    // save connection state in a variable
    gOnline = connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;

    // if online now, try to connect to web services proxy
    if (gOnline==true)
    {
        myBackgroundTask("start");
    }
    else
    {
        myBackgroundTask("stop");
    }
};
```

### Parameters

*connected* Boolean value: true if user is connected; false if offline.

### Returns

Nothing.

## Description

Agent, application, or pod event handler; called by the respective shell when the connection status (online or offline) changes. The shell does not automatically check for connectivity; it simply follows the user setting made in the File menu (either work online or work offline) or when the user selects the network icon (lightning bolt). To determine if the user is online when the application first loads, use the `isConnected()` method. You can then manually trigger your own `onNetworkChange()` handler so that your contained scripts run. That is, Central only triggers `onNetworkChange()` when users manually change their connection status. To check the status using a script, use the `isConnected()` method.

Central can't automatically recognize whether a computer is connected to the Internet; it honors the user's setting.

A best practice is to first check the current status (using `isConnected()`) and save that status in a variable. Do not attempt online access when the status is `false`. When `onNetworkChange()` reports `true` (in other words, when going back online), reestablish any background network access, connecting to data, and updating as needed. For example, call a `setInterval` function to periodically call a web service and get up-to-date information. For more information on using the agent to manage data, see [Chapter 2, "Understanding the Macromedia Central Environment," on page 19](#).

## Pod.onNoticeEvent()

### Availability

Macromedia Central.

### Usage

```
// Handle a change to an existing Notice from this app
onNoticeEvent = function ( event, noticeData, initialData )
{
    // trace the properties contained in this notice
    trace("event.type="+event.type);
    trace("optional data from issuing app "+initialData);
    for (var i in noticeData){
        trace("noticeData."+i+"="+noticeData[i]);
    }

    // respond according to the event type
    switch (event.type){
        case "close":
            message_txt.text="notice id "+noticeData.id+" was closed";
            break;

        case "engage":
            message_txt.text="you engaged "+noticeData.description;
            break;

        case "timeout":
```

```

        message_txt.text="elapsed time reached "+noticeData.timeout;
        break;

    case "remove":
        message_txt.text="removed the notice named "+noticeData.name;
        break;
    }
    // remove this notice from the list of notices we're maintaining
    if(event.type!="engage"){
        myRefreshListOfNotices();
    }
};

```

## Parameters

*event* An object containing one string element, *type*, that provides the reason for the notice's dismissal. The *type* element has one of the following values:

Value	Description
close	Closed by the user by selecting the close box in the notice list.
engage	Closed by the user by selecting the engage button as in the notice detail.
timeout	Dismissed by Central because the notice has timed out.
remove	Dismissed by the application through a call to <code>removeNotice()</code> .

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
id	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
name	A string to be displayed in the notice's title bar. If not specified, the default value of <i>name</i> is: <i>application name</i> Notice
description	Longer string to be displayed in the notice's body. The default is an empty string.
timeout	A number that specifies the seconds after which the notice should be automatically dismissed. Set <i>timeout</i> to 0 to create a notice that never times out.
alert	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
engageString	A short string; displayed on the engage button. If not specified, no engage button appears.
navigate	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
unread	A Boolean value that indicates whether the notice has been viewed by the user.



*initialData* Any data type specifying application-specific data passed at the time you call `addNotice()`.

### Returns

Nothing.

### Description

Agent, application, or pod event handler; invoked in an agent, application, or pod when a notice created by your application is dismissed. Any of the following events will trigger `onNoticeEvent()`: when the user clicks the close box or clicks the engage text, if the notice times out, or the notice is removed programmatically through the `removeNotice()` method.

The specific values contained in the *noticeData* and *initialData* come from the initial call to `addNotice()`. The `onNoticeEvent()` is not triggered unless your application first creates a notice through `addNotice()`. For an example of how to create a notice and add it, see [AgentManager.addNotice\(\) on page 146](#).

A common use of this method is to include more detail about a notice in a related window. Presumably your user wanted the notice. The `onNoticeEvent()` handler is your opportunity to give the user further details. By passing application-specific data through *initialData*, the application can show the correct item related to a notice (for example, a stock chart view related to a notice about that stock).

## Pod.onPositionChange()

### Availability

Macromedia Central.

### Usage

```
onPositionChange = function (newPosition)
{
    //save position index value in a variable
    gPosition = newPosition;
}
```

### Example

```
OnPositionChange(newPosition) {
    if (newPosition ==0) //we are at the top of the console!!
        atTop = true;
}
```

### Parameters

*newPosition* Integer; represents the zero-based index of the pod position from the top of the Console.

### Returns

Nothing.

## Description

Called by the Console when the user drags a pod to a new position, when the user adds a pod to the Console, or when the user removes another pod above the current pod.

## Pod.onSelectedItem()

### Availability

Macromedia Central.

### Usage

```
onSelectedItem=function(data)
{
    //process the data received
};
```

### Example

```
//RECEIVER APPLICATION:

//place the following in the product.xml's pod and/or application section
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
    <type>any</type>
</supportedTypes>

//this method will populate an MListBox component
onSelectedItem=function(data)
{
    //prepare a List component to populate
    myListComponent.removeAll();

    for(var i=0;i<data.length;i++)
    {
        //make sure we treat the data as a selectedItem (not XML)
        var thisItem=data[i].asSelectedItem();
        myListComponent.addItem(thisItem.name, thisItem.description);
    }
};

//SENDER APPLICATION:

//this part of the code shows how an application can prepare data to send
onActivate=function(shell){
    //set a variable to reference the Shell or Console
    gShell=shell;

    send_btn.onPress=function
    {
        //prepare the data as an array of two selectedItem items
        var dataToSend=new Array();

        var item1=new SelectedItem("http://www.mysite.com/ns#", "aType");
        item1.name="name one";
        item1.description="this is the description for item 1";
```

```

dataToSend.push(item1);

var item2=new SelectedItem("http://www.mysite.com/ns#", "aType");
item2.name="name two";
item2.description="this is the description for item 2";
dataToSend.push(item2);

//with the items prepared, set the data array and a prompt
gShell.setSelectedItem([item1, item2], "blast two items!");
    };
};

```

## Parameters

*data* An array of instances of the SelectedItem ActionScript structure or an array of XML objects.

## Returns

Nothing.

## Description

Application and pod event handler; called by the shell when data is arriving in your application. For this to happen, another application has to first make a selection of a data type that your application supports. Then a user can manually select your application from the Blast menu or, when the Auto Blast option is enabled, it triggers immediately. Although there are several steps involved in defining the supported data types (in the product.xml file) and preparing a selection to broadcast (in the sending application), `onSelectedItem()` is where you define how your application responds when others send data to it using the Blast feature.

The selection is local to each shell window, so an application in one shell window can set the selected item without destroying the selected item in another shell window.

**Note:** All pods that can receive Blast data will receive it when the user selects the Edit > Blast > All On Screen menu option. Only applications that can receive data show up by name and are listed in the Edit > Blast menu. However, to receive data, pods must still register types that they support in the product.xml file.

Your application needs to handle the received data as an array full of ActionScript structures or an array of XML data. The following two functions are built into Central to make it easy to treat the received data in the form you prefer. (The application that receives the data may not detect in which form it was sent.)

```

asXML(); // returns XML object
asSelectedItem(); // returns SelectedItem object

```

Regardless of whether the data contains structures or XML, you can turn it into the form you want. Macromedia recommends using the `ASXML()` structure approach for sending and receiving; this eliminates any extra overhead from converting to or from XML, which is naturally more verbose and thus less efficient. You can also use the `asXML()` function to convert data into its XML form for communicating with external sources, debugging, and so on. Similarly, you can use the `asSelectedItem()` function to convert any old XML into `SelectedItem` object instances. Remember, though, to convert any old XML into a `SelectedItem` object properly; the `schemaType` field must be set in order to send XML data directly. For more information on the Blast feature, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Pod.setBaseTabIndex()

### Availability

Macromedia Central.

### Usage

```
setBaseTabIndex=function( newIndex )
{
    //set up controls' tabIndex vales
}
```

### Example

```
//this example shows a complete strategy to keep tab settings refreshed
onActivate = function(console, podID, viewerID, position, baseTabIndex)
{
    //set a variable to reference the shell
    gConsole = shell;

    //set an array of tab-able controls
    gControls=[username_txt, password_txt, continue_btn];

    //call our own setBaseTabIndex once initially
    this.setBaseTabIndex(baseTabIndex);
};

//refresh the controls in the pod
setBaseTabIndex=function(newIndex)
{
    for(var i=0;i<gControls.length;i++)
    {
        gControls[i].tabIndex=newIndex+i;
    }
    gMaxTabUsed=newIndex+i;
};

//set up handler to report back to the Console
getLastTabIndex=function()
{
    return gMaxTabUsed;
};
```

## Parameters

*newIndex* Integer; represents the Central next available tab index. Use this index as your base tab index.

## Returns

Nothing.

## Description

Pod method; called by the Console to notify your pod that its `baseTabIndex` has changed and therefore you need to reset the `tabIndex` values for controls such as text and buttons. Setting `tabIndex` values using the `baseTabIndex` received in your `onActivate()` code until pods are added or otherwise rearranged (say, by the user repositioning them) is important, because the console calls `setBaseTabIndex()` as a way of saying that it's time to update your tab indexes.

The example code shows a complete and fairly workable strategy. It may seem like a lot of work, but remember, your pod was loaded into a single SWF file (the Console window) along with other pods and notices.

## RegExp object

**ActionScript Class Name** `mx.central.RegExp`

The `RegExp` object provides support for regular expression evaluation from ActionScript. You can use the `RegExp` object whenever you need to match ActionScript strings. For example, the expression `"hunt?"` can be used to find all instances of `"hunt"`, `"hunted"`, `"hunting"`, and `"hunter"`. The expression `"cp *.xml ../"` can be used to find all files that have filenames ending in `".xml"`.

Regular expressions constitute a large area of functionality that is documented at length in many locations on the Internet. The implementation of regular expressions in Central is nearly identical to the ECMA-262 Edition 4 specification (see [www.mozilla.org/js/language/es4/index.html](http://www.mozilla.org/js/language/es4/index.html)), with the exception that the Central implementation does not support literals. For more information about using regular expressions in Central, Macromedia has an online article at [www.macromedia.com/devnet/central/articles/regex\\_04.html](http://www.macromedia.com/devnet/central/articles/regex_04.html).

## Method summary for the RegExp object

Method	Description
<a href="#">RegExp.exec()</a>	Perform search (ECMAScript name).
<a href="#">RegExp.match()</a>	Same as <code>exec</code> (name used by JavaScript).
<a href="#">RegExp.test()</a>	Equivalent to <code>exec(string) != null</code> .
<a href="#">RegExp.replace()</a>	For every instance where the <code>RegExp</code> object matches in <code>string</code> , replace with <code>repl</code> .

## Property summary for the RegExp object

Property	Description
<a href="#">RegExp.dotall</a>	Whether dot (.) matches new lines.
<a href="#">RegExp.extended</a>	Whether in extended mode.
<a href="#">RegExp.global</a>	Whether global search is on.
<a href="#">RegExp.ignoreCase</a>	Whether case is being ignored.
<a href="#">RegExp.lastIndex</a>	The index of the last match.
<a href="#">RegExp.multiline</a>	Whether multiline is on.
<a href="#">RegExp.source</a>	The source string of the last match.

## Regular Expression Syntax

The following tables list the special character types supported in this RegExp API.

### Basic Support

Token	Description
<code>^</code>	Beginning of line/string (see <code>multiline</code> property)
<code>\$</code>	End of line/string (see <code>multiline</code> property)
<code>.</code>	Match any character
<code>[</code>	Range of characters
<code>re re</code>	Choice
<code>re*</code>	Zero or more
<code>re+</code>	One or more
<code>re?</code>	Zero or one
<code>\</code>	Escape character
<code>[]</code>	Range specifier (start-end) and/or group of characters

### Extensions

Token	Description
<code>re*?</code>	Zero or more nongreedy
<code>re+?</code>	One or more nongreedy
<code>re??</code>	Zero or one nongreedy
<code>re{}</code>	A numbered or ranged multiplier
<code>re{ }?</code>	(Nongreedy)
<code>\b</code>	Either end of word
<code>\B</code>	Not either end of word

Token	Description
\d	[0-9]
\D	[^0-9]
\s	Whitespace
\S	Not whitespace
\w	Alphanumeric and _
\W	Not alphanumeric or _
(re)	Group
(?= re )	Look ahead
(?! re )	Negative look ahead
(?: re )	Noncapturing group (for overriding precedence rules)
(?P<name>re)	Named group (from <i>Python</i> )

## Constructor for the RegExp object

When creating a new `RegExp` object, you pass in a string and any flags, as follows:

```
re = new RegExp("My String", myFlag);
```

For example, to search for an HTML string, you define the string and create a new `RegExp` object. The `result` variable then contains the match, as the following example shows:

```
str = "This is an HTML <b>string</b>";
re = new RegExp("<b>(.*?)</b>");
result = re.match(str); // result[0] == "string"
```

Possible flags are:

- `i` - ignoreCase
- `s` - dotall
- `m` - multiline
- `x` - extended
- `g` - global

## RegExp.dotall

### Availability

Macromedia Central.

### Usage

```
RegExp.dotall
```

### Description

`RegExp` property; specifies whether dot (`.`) matches new lines. Use the `s` flag when constructing this `RegExp` object to set `dotall = true`.

## RegExp.exec()

### Availability

Macromedia Central.

### Usage

`RegExp.exec(string)`

### Parameters

*string* String to search.

### Returns

Return an object with the following properties:

Property	Description
index	Index into string of match
input	Original string
[0]	Matched string
[1], ..., [n]	Parenthesized groups
<name>...	Named groups (see <a href="#">"Regular Expression Syntax" on page 326</a> )

### Description

RegExp method; performs search (ECMAScript name).

## RegExp.extended

### Availability

Macromedia Central.

### Usage

`RegExp.extended`

### Description

RegExp property; specifies whether in extended mode. Use the `x` flag when constructing this RegExp object to set `extended = true`. Read-only property.

When a RegExp object is in extended mode, whitespace characters in the constructor string are ignored. This is done to allow more readable constructors.

### Example

The following example illustrates the different ways to construct a RegExp object. The second constructor uses the `x` flag, causing the whitespaces in the string to be ignored.

```
// matches either xxx-xxx-xxxx or (xxx)xxx-xxxx
var rePhonePattern = new RegExp("^(?:\\d{3}-\\d{3}-\\d{4}|\\(\\d{3}\\)\\d{3}-\\d{4})$");
```



```
// more readable as:  
var rePhonePattern2= new RegExp("^(?:\\d{3}-\\d{3}-\\d{4} |  
  \\(\\d{3}\\)\\s?\\d{3}-\\d{4})$", "x");
```

## RegExp.global

### Availability

Macromedia Central.

### Usage

RegExp.global

### Description

RegExp property; specifies whether global search is on. Read-only property. Use the *g* flag when constructing this RegExp object to set `global=true`. When `global=true`, the `lastIndex` property is set after a match is found. The next time a match is requested the regular expression engine starts from the `lastIndex` position in the string.

## RegExp.ignoreCase

### Availability

Macromedia Central.

### Usage

RegExp.ignoreCase

### Description

RegExp property; specifies whether case is being ignored. Read-only property. Use the *i* flag when constructing this RegExp object to set `ignoreCase = true`.

## RegExp.lastIndex

### Availability

Macromedia Central.

### Usage

RegExp.lastIndex

### Description

RegExp property; the index of the last match. Read-only property. Will always be 0 unless `global=true` for this RegExp object.

## RegExp.match()

### Availability

Macromedia Central.

### Usage

RegExp.match(*string*)

## Parameters

*string* String to match.

## Returns

Returns an object with the following properties:

Property	Description
index	Index into string of match
input	Original string
[0]	Matched string
[1], ..., [n]	Parenthesized groups
<name>...	Named groups (see <a href="#">“Regular Expression Syntax” on page 326</a> )

## Description

RegExp method; same as the `exec` method (name used by JavaScript).

## RegExp.multiline

### Availability

Macromedia Central.

### Usage

`RegExp.multiline`

### Description

RegExp property; specifies whether multiline is on, allowing a value to be more than one line. Read-only property. Use the `m` flag when constructing this RegExp object to set `multiline = true`.

## RegExp.replace()

### Availability

Macromedia Central.

### Usage

`RegExp.replace(string, repl)`

### Parameters

*string* String to match.

*repl* String with which to replace the matched string.

### Returns

Nothing.

### Description

RegExp method; for every instance where the RegExp matches in *string*, replace with *repl*.

## RegExp.source

### Availability

Macromedia Central.

### Usage

RegExp.source

### Description

RegExp property; source string of the last match.

## RegExp.test()

### Availability

Macromedia Central.

### Usage

RegExp.test(*string*)

### Parameters

*string* String to test for.

### Returns

Boolean value: true if the string is matched; otherwise, false.

### Description

RegExp method; equivalent to `exec(string) != null`.

## RPC object

**ActionScript Class Name** mx.central.services.RPC

RPC callback object; this is the object type returned each time the `createCall()` method is called on an `RPCFactory` object. The RPC callback object receives events when results or faults are returned from the RPC method called with `createCall()`. That is, not only does `createCall()` return an instance of the RPC object, but it also invokes the remote method. Callbacks defined for the RPC object instance are how you handle the results.

```
myRPCObj = myRPCFactoryObject.createCall(myMethod [, argument1, ...argument2]);
```

For more information, see [“RPCFactory object” on page 334](#).

## Method summary for the RPC object

Method	Description
None.	

## Property summary for the RPC object

Property	Description
<code>RPC.response</code>	The raw XML version of the results returned from <code>createCall()</code> .

## Event handler summary for the RPC object

Event handler	Description
<code>RPC.onFault()</code>	Called by Central when the method called with <code>createCall()</code> generates an error.
<code>RPC.onResult()</code>	Called by Central when the method called with <code>createCall()</code> returns a result.

## RPC.onFault()

### Availability

Macromedia Central.

### Usage

```
// create an RPCFactory object
myRPCFactory = new RPCFactory(RPCUrl);

// create an RPC object and trigger the remote call
myRPCObject = myRPCFactory.createCall(myMethod, argument1);

// handles any errors returned
myRPCObject.onFault = function(fault)
{
    // save the fault in the gSavedFaultStructure homemade variable
    gSavedFaultStructure = fault;
};
```

### Parameters

*fault* An ActionScript object with properties that map to the XML-RPC structure type of the fault.

### Returns

Nothing.

### Description

RPC callback object event handler; Central calls this method when the `createCall()` method has failed and returns an error. You need to write this handler for the RPC instance returned when you invoke `createCall()` on the RPCFactory instance. The `fault` parameter is the ActionScript object version of an XML fault structure.

## RPC.onResult()

### Availability

Macromedia Central.

### Usage

```
// create an RPCFactory object
myRPCFactory = new RPCFactory(RPCUrl);

// create an RPC object and trigger the remote call
myRPCObject = myRPCFactory.createCall(myMethod, argument1);

// handle the results
myRPCObject.onResult = function(result)
{
    // display the result in an onscreen text field
    myResults_txt.text = result;
};
```

### Parameters

*result* A decoded ActionScript object version of the XML result returned by an RPC web service method called with `createCall()`.

### Returns

Nothing.

### Description

RPC callback object event handler; Central calls this method when the `createCall()` method successfully returns a result. You need to write this handler for the RPC instance returned when you invoke `createCall()` on the `RPCFactory` instance. The *result* parameter is the ActionScript object version of an XML fault structure. See [“RPC.response” on page 333](#).

## RPC.response

### Availability

Macromedia Central.

### Usage

```
// create an RPCFactory object
myRPCFactory = new RPCFactory(RPCUrl);

// create an RPC object and trigger the remote call
myRPCObject = myRPCFactory.createCall(myMethod, argument1);

myRPCObject.onResult = function(result)
{
    theRawXML = myRPCObject.response;
};
```

## Description

RPC callback object property; contains the raw XML form of the response to the most recent web service method call made using `createCall()`. Use `myRPCCallbackObj.onResult()` to get the corresponding `ActionScript` version of the results of the request. As the example shows, you should wait until the `onResult()` callback is triggered before attempting to access the `response` property.

## RPCFactory object

**ActionScript Class Name** `mx.central.services.RPCFactory`

This object contains a parsed version of the entire XML-RPC file you specify in the *RPCUrl* parameter. You can then use this `RPCFactory` instance to create calls (based on methods defined in the XML-RPC file). That is, with an instance of the `RPCFactory` object, you can invoke a single method, `createCall()`, that both triggers a remote method and returns an instance of the `RPC` object. You then define methods on the `RPC` object to handle results and errors. For more information, see [“RPC object” on page 331](#).

## Method summary for the RPCFactory object

Method	Description
<code>RPCFactory.createCall()</code>	Invokes a remote method (in your <code>RPCFactory</code> instance's XML-RPC file) and returns an <code>RPC</code> object for which you can define callbacks that handle the results and faults.

## Property summary for the RPCFactory object

Property	Description
None.	

## Event handler summary for the RPCFactory object

Method	Description
None.	

## Constructor for the RPCFactory object

### Availability

Macromedia Central.

### Usage

```
myRPCFactoryInstance = new RPCFactory(RPCUrl);
```

### Parameters

*RPCUrl* The location where your `RPC-XML` file resides. This file contains the remote method declarations that Central then parses.

## Returns

Nothing.

## Description

Constructor; creates an `RPCFactory` object. Creates an instance of the `RPCFactory` object (XML-RPC). This object contains a parsed version of the entire XML-RPC file you specify in the `RPCUrl` parameter. You can then use this `RPCFactory` instance to create calls (based on methods defined in the XML-RPC file). That is, with an instance of the `RPCFactory` object, you can invoke a single method, `createCall()`, that both triggers a remote method and returns an instance of the `RPC` object. You then define methods on the `RPC` object to handle results and errors. For more information, see [“RPC object” on page 331](#).

## `RPCFactory.createCall()`

### Availability

Macromedia Central.

### Usage

```
myRPCObj = myRPCFactoryObject.createCall(myMethod [, argument1, ...argument2])
```

### Parameters

The parameters that are required depend on the `RPC` method being called. Typically, the first parameter is the method name, which is followed by any arguments that the method requires.

## Returns

An `RPC` callback object. This object can receive the results of the `RPC` method or any faults that the method generates. To receive these events, implement the `RPC.onResult()` and `RPC.onFault()` handlers. See [“RPC.onResult\(\)” on page 333](#) and [“RPC.onFault\(\)” on page 332](#).

## Description

`RPCFactory` object method; invokes a remote `RPC` method defined in the XML-RPC file supplied when the `RPCFactory` object is created ( `new RPCFactory(RPCUrl)` ). Because the call is asynchronous, `createCall()` returns an object for which you need to define an `onResult()` callback to handle results and an `onFault()` callback to handle errors.

## SelectedItem object

**ActionScript Class Name**   `mx.central.data.SelectedItem`

Creating a `SelectedItem` instance using `new SelectedItem()` is the first step for sending Blast data. As the example shows, you need to populate the properties within your instance with values. Finally, when you're ready to make that data available to the Blast menu, use the `setSelectedItem()` method. Populating this object is only the end of the “sending” side of the Blast feature. When the user chooses to send data using the Blast feature, the application on the “receiving” side needs an `onSelectedItem()` callback defined that handles the data received. For more about setting the selection or about receiving the Blast data, see [“Using XML objects to send data” on page 112](#).

When you decide to make data available through the Blast feature to other applications, you need to also decide which data type (within a particular namespace) to make available. The developer of the application intending to receive your data must specify the same data type in their `supportedTypes` tag (in the `product.xml` file). Although an application can say it supports any data type after that data is received, the data will need to be parsed. Since it is practically impossible to write a parsing script that can intelligently make sense of any data type, you should either select and use a common data type (such as one listed in the `CentralData.xsd` file) or publish a schema for the data types you're going to send using the Blast feature. Not only does a custom data type require additional design time on your part (you may be “reinventing the wheel”), but the other developers will need to know about your schema ahead of time. If you intend to only support sending data to and from your own products using the Blast feature, this is not an issue. However, the Blast feature becomes more powerful when disparate applications can cooperate with each other.

For more about the `supportedTypes` tag, see [“Registering supported data types in the product.xml file” on page 109](#).

For more information about the Blast feature, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Constructor for the `SelectedItem` object

### Availability

Macromedia Central.

### Usage

```
mySelectedItemInstance = new SelectedItem(nameSpace, dataType)
```

### Example

```
// RECEIVER APPLICATION:

// place the following in the product.xml file's pod and/or application section
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
  <type>any</type>
</supportedTypes>

// this method will populate an MListBox component
onSelectedItem=function(data)
{
  // prepare a List component to populate
  myListComponent.removeAll();

  for(var i=0;i<data.length;i++)
  {
    // make sure we treat the data as a selectedItem (not XML)
    var thisItem=data[i].asSelectedItem();
    myListComponent.addItem(thisItem.name, thisItem.description);
  }
};
```



```
// SENDER APPLICATION:

// this part of the code shows how an application can prepare data to send
onActivate=function(shell){
    // set a variable to reference the Shell or Console
    gShell=shell;

    send_btn.onPress=function()
    {
        // prepare the data as an array of two selectedItem items
        var dataToSend=new Array();

        var item1=new SelectedItem("http://www.mysite.com/ns#", "aType");
        item1.name="name one";
        item1.description="this is the description for item 1";
        dataToSend.push(item1);

        var item2=new SelectedItem("http://www.mysite.com/ns#", "aType");
        item2.name="name two";
        item2.description="this is the description for item 2";
        dataToSend.push(item2);

        // with the items prepared, set the data array and a prompt
        gShell.setSelectedItem([item1, item2], "blast two items!");
    };
};
```

## Parameters

*nameSpace* A string that uniquely identifies all the data types you are using. Traditionally, this is a URL (containing an XSD schema) that declares how the data type is structured.

*dataType* A string that identifies what data type you want to cast. This string determines how you can populate the data, as well as what other applications can receive it when transmitted through the Blast feature. (Only applications that support this data type can receive the Blast data.)

## Returns

[SelectedItem object](#); an instance returned that you can use to populate with values and then make available for sending with the Blast feature by using the `setSelectedItem()` method.

## Description

Constructor function; used to create an instance in the form of a SelectedItem object. You need to specify a unique *nameSpace* in addition to the *dataType* in case another developer designs a data type with the same name as yours. Including a namespace means that two developers could both have a different implementation of the same named data type with no conflict. For instance, “developer 1 thing” and “developer 2 thing” are both “things,” but they’re different. Macromedia has defined some basic data types to get you started; these are defined in the first version of the CentralData.xsd schema.

Also, the following two functions are built into Central to make it easy to treat the received data in the form you prefer. (The application that receives the data may not know in which form it was sent.)

```
asXML(); // returns XML object
```

```
asSelectedItem(); // returns SelectedItem object
```

Regardless of whether the data contains structures or XML, you can turn it into the form you want. Macromedia recommends using the `ActionScript` structure approach for sending and receiving; this eliminates any extra overhead from converting to or from XML, which is naturally more verbose and thus less efficient. The `asXML()` function could also be used to convert data into its XML form for communicating with external sources, debugging, and so on. Similarly, the `asSelectedItem()` function could be used to convert any old XML into `SelectedItem` object instances. Remember, to convert any old XML into a `SelectedItem` object properly, the `schemaType` field has to be set in order to send XML data directly.

## Shell object

**ActionScript Class Name** `mx.central.Shell`

Applications communicate with the Central environment through the Shell object. The shell is to your application what the Console is to your Pods and what the `AgentManager` is to your Agent. That is, you can think of the shell as the window that holds your application. Your application receives a reference to the shell as the first parameter in the `onActivate()` method. That reference is used whenever you want to access any methods in the Shell object. If you want your application to communicate directly with your pods or agent, you should use your own implementation of the `LocalConnection` object or the `Central LCService` class developed specifically for this purpose.

The following methods are implemented by the shell, and are called by your application by using a reference to the shell. (That is, you always replace *Shell* with a variable that contains the reference to the shell received in your `onActivate()` handler.)

### Method summary for the Shell object

Method	Description
<code>Shell.addNotice()</code>	Called by your application to create a new notice.
<code>Shell.addPod()</code>	Called by your application to make a pod available in the Console. (The pod doesn't become visible until the user opens it or you call <code>viewPod()</code> .)
<code>Shell.addToLocalInternetCache()</code>	Called by your application to add a URL to the local Internet cache.
<code>Shell.editLocationDialog()</code>	Called by your application to open the Edit Location dialog box in the same way as if the user manually selects Edit Locations from the Location pop-up menu in the Identity & Location section of the general preferences. This gives the user the opportunity to make changes to their location settings.

Method	Description
<code>Shell.getAgent()</code>	Called by your application to access various properties of the agent, such as whether it's currently running.
<code>Shell.getBounds()</code>	Called by your application to ascertain the current window size of your application. (An <code>ActionScript</code> object with <code>height</code> and <code>width</code> is returned.)
<code>Shell.getNotices()</code> <code>Shell.getNotices()</code>	Returns an array of <code>ActionScript</code> objects, each containing details about the notices created by your application that are still present.
<code>Shell.getPods()</code>	Returns an array of <code>ActionScript</code> objects, one for each pod available to your application (as listed in the <code>product.xml</code> file or created using <code>addPod()</code> ) and each containing details about that pod.
<code>Shell.getPreferences()</code>	Gets the user preferences that have been exposed to your application.
<code>Shell.getSelectedItem()</code>	Returns the most recent <code>SelectedItem</code> that you created using <code>setSelectedItem()</code> . Populating a <code>SelectedItem</code> with data is the first step in making that data available to other applications through the Blast feature.
<code>Shell.getViewedApplications()</code>	Returns an array of <code>ActionScript</code> objects, each containing details about each <code>Shell</code> instance (that is, separate window) currently running your application.
<code>Shell.getViewedPods()</code>	Returns an array of <code>ActionScript</code> objects, each containing details about the pod instances currently arranged in the Console.
<code>Shell.inLocalInternetCache()</code>	Called by your application when it wants to check whether a URL is in the local Internet cache. (Returns <code>true</code> if it is, <code>false</code> otherwise.)
<code>Shell.isConnected()</code>	Called by your application to determine current network status.
<code>Shell.isConsoleOpen()</code>	Called by your application to determine if the console is currently open.
<code>Shell.isPurchased()</code>	Reserved. Attribute currently unavailable.
<code>Shell.newLocationDialog()</code>	Called by your application to open the New Location dialog box in the same way as if the user manually selects New Locations from the Location pop-up menu in the Identity & Location section of the general preferences. After the user names the new location, the standard preference dialog box appears. You also have the option to tag specific fields as required, although the user can always cancel the operation.
<code>Shell.removeFromLocalInternetCache()</code>	Called by your application to remove a specific URL (such as an image file) from the local Internet cache.

Method	Description
<code>Shell.removeNotice()</code>	Called by your application when you want to remove a notice using the notice ID returned at the time that the notice was added.
<code>Shell.removePod()</code>	Called by your application when you want to remove a pod using the pod ID returned when the pod was added. (Unlike when a user closes a pod, this makes the pod no longer accessible.)
<code>Shell.requestPayment()</code>	Reserved. Attribute currently unavailable.
<code>Shell.requestSizeChange()</code>	Called by your application to request a change to the size of the application window (height and width). Sometimes the user's screen size prevents you from resizing.
<code>Shell.setProgress()</code>	Called by your application to change the Central built-in progress bar at the bottom of the shell. This method lets you do one of three things: you can display a specific percentage, display the indeterminate indicator (a continuous barber pole animation), or remove the progress bar from view.
<code>Shell.setSelectedItem()</code>	Called by your application to notify the shell that you have prepared new data (by way of the user making a selection, for example) that you want to make available to other applications through the Blast feature.
<code>Shell.setStatus()</code>	Called by your application to set a current message in the status area at the bottom of the shell.
<code>Shell.startAgent()</code>	Called by your application to start the agent associated with this application (in the product.xml file).
<code>Shell.stopAgent()</code>	Called by a pod to stop the agent associated with this application (in the product.xml file).
<code>Shell.validateActivationKey()</code>	Reserved. Interface currently unavailable.
<code>Shell.viewPod()</code>	Calling this function makes the specified pod viewable in the top viewer (that is, the uppermost tile) of the Console. (This method requires that the specified pod is first identified in your product.xml file or created using <code>addPod()</code> .)

## Property summary for the Shell object

Property	Description
None.	

## Event handler summary for the Shell object

Event handler	Description
None.	

## Shell.addNotice()

### Availability

Macromedia Central.

### Usage

```
noticeID=shellReference.addNotice(noticeData [,initialData])
```

### Example

```
// This example function adds a notice based on parameters received
// You could use it as follows:
// var thisID=postStockNotice("MACR", 20, "a description", true);
// myListOfNotices.push(thisID);

postStockNotice=function(ticker, price, ruleDescription, alert)
{
    // Creates a new notice object
    var noticeData = new Object();
    noticeData.name = ticker + " " + price;
    noticeData.description = ruleDescription;
    noticeData.alert = alert;
    noticeData.engageString = "show";
    // add noticeData using a reference to gShell (received in onActivate)
    var noticeID = gShell.addNotice(noticeData, {ticker: ticker});

    // return the ID of this notice for future reference
    return noticeID;
}
```

### Parameters

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<i>id</i>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<i>name</i>	A string to be displayed in the notice's title bar. If not specified, the default value of <i>name</i> is: <i>application name</i> Notice
<i>description</i>	Longer string to be displayed in the notice's body. The default is an empty string.
<i>timeout</i>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <i>timeout</i> to 0 to create a notice that never times out.
<i>alert</i>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<i>engageString</i>	A short string; displayed on the engage button. If not specified, no engage button appears.

Property	Description
navigate	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
unread	A Boolean value that indicates whether the notice has been viewed by the user.

*initialData* Arbitrary application-specific data of any type. This data is received as the third parameter in an `onNoticeEvent` callback.

## Returns

`NoticeID` used to refer to this notice in later calls.

## Description

`AgentManager`, `Console`, or `Shell` method; triggered by an agent, pod, or application, respectively, to create a new notice. You need a reference to the appropriate shell (returned as the first parameter in the `onActivate` event) to which you trigger this method. The examples use `gShell` with the assumption that that variable was set by `onActivate`. For more information on getting a reference to the shell, see `Agent.onActivate`, `Application.onActivate`, or `Pod.onActivate`.

It's good practice to store some identifying information in the optional `initialData` parameter when adding a notice. When the user engages the notice, the identifying information is received in the `onNoticeEvent` event.

Also, it's often better to update a notice instead of adding a new one. You update a notice deleting the old one and replacing it with a new one. This requires you to keep track of the notices as you create them.

## Shell.addPod()

### Availability

Macromedia Central.

### Usage

```
podID=shellReference.addPod(podData)
```

### Example

```
// Create a pod when your application loads
onActivate = function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    var gShell = shell;

    // trigger a homemade function that creates a named pod
    var days=["sun","mon","tue","wed","thu","fri","sat"];
    var dayName=days[new Date().getDay()];
    createPod(dayName+"_pod");
}
```

```

}

// creates and opens a uniquely named pod based on a specific class
createPod = function (theName)
{
    // Create a new pod and populate it
    var podData = new Object();

    // Set the name that displays on the pod itself
    podData.name = theName;

    // This value must be the same as the <podClass name="name"> tag
    podData.className = "calendarClass";

    // Set an initial value to keep with the pod
    podData.initialData = new Date();

    // Add the pod and save a reference to it
    var thisPodID = gShell.addPod( podData );

    // Use the agentManager reference to view this pod in the console
    gShell.viewPod(thisPodID);

};

```

## Parameters

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A *podData* object includes the following properties:

Property	Description
id	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
name	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.

Property	Description
<code>className</code>	<p>A string that specifies the name of the class, as defined in the product.xml file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your <code>pod</code>. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the product.xml file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<code>height</code>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<code>src</code>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the product.xml file's <code>pod</code> tag or <code>podClass</code> tag.)
<code>enabled</code>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<code>appid</code>	A number set by Central to associate a pod with your application.
<code>supportedTypes</code>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the product.xml file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<code>initialData</code>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

## Returns

`podID`; Number set by Central and representing the unique identifier of the pod instance.

## Description

AgentManager, Shell, or Console method; called by an agent, application, or pod, respectively, to add a pod to the Console. The `addPod()` method only makes a new pod instance available, and `viewPod()` actually makes the pod appear (as though the user physically selected it from the Console's pod pop-up menu). You need to use the `podID` returned from the `addPod()` method to trigger the `viewPod()` method.



The hierarchy of application, pod, and podClass is important. As long as your application has at least one pod defined in the product.xml file, the user can instantiate multiple pods in the Console. While the Console only lists applications with pods available, it won't list your application more than once. This is true even if you include multiple `pod` tags (in the product.xml file) or if you create multiple instances of a pod (either with `addPod()` or through the product.xml file). If an application has more than one pod available, the user will see that choice in a secondary pop-up menu inside the pod itself (next to where the pod's name appears).

Generally speaking, the user has the ultimate control over how pods are presented. However, through `addPod()`, your application can make more pods available, and through `viewPod()` added pods can be displayed. There are also methods to determine which pods are available and which are currently being viewed (`getPods()` and `getViewedPods()` respectively). In addition, with a `podID` you can use the `removePod()` method to eliminate a particular pod. However, this is not the same as a user closing a pod—`removePod()` makes the pod unavailable. There are lots of options available, but keep in mind that the goal is to provide the user with intuitive tools that provide flexibility during development.

## Shell.addToLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
shellReference.addToLocalInternetCache(url [, bOverwrite, expiration])
```

### Example

```
// this example adds a JPG to the cache, loads it, then checks if successful
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    var theFile="http://www.mysite.com/images/photo.jpg"
    // add it to the cache
    gShell.addToLocalInternetCache(theFile);
    someClipInstance.loadMovie(theFile);

    // check that the disk quota wasn't exceeded
    if(gShell.inLocalInternetCache(theFile)==true)
    {
        // trigger homemade function to explain the image wasn't downloaded
        myAlertFunction("The photo won't be available when offline");
    }

};
```

### Parameters

*url*   String; a fully qualified URL where the file to be cached resides.

*bOverwrite* Optional parameter; a Boolean value that indicates whether to overwrite preexisting files of the same name. If the value of *bOverwrite* is `true` and the file indicated as the *url* value is already in the cache, Central overwrites the file. The default value for *bOverwrite* is `false`.

*expiration* Optional parameter; either a `Date` object or a number. This value indicates when the locally cached file will be considered out of date. If you provide a `Date` object for this value, Central considers the file current until the date indicated. If you do not include an expiration date, the default expiration for any cached file is three days. If you provide a number for this value, Central considers the file current for that number of days.

## Returns

None.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to add a URL to the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the cache rather than from the web, enabling products to use data even when the user is offline. To ensure your application loads the URL from the Internet, first call `addToLocalInternetCache()` with the *bOverwrite* parameter set to `true`.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

Usually, you'll call the `addtoInternetCache()` method before loading an image or data file. Regardless of how long the download takes, you can immediately call a command such as `loadMovie()` to load the same file. Adding to the cache simply means that file is saved on the user's hard disk. Although the `addtoIntenetCache()` method does download the file, it primarily adds URLs to a list from which Central always checks before attempting to download from the Internet.

Files do not expire when a user is offline. Similarly, once a file is expired it isn't automatically removed from the cache. Rather, subsequent attempts to load that URL attempt to access the Internet unless the user is offline. If the user is online and the `inLocalInternetCache()` method is called, the file in question will be removed from the cache if it's expired. If the user is online and the `addToInternetCache()` method is called, the file in question will be overwritten if it's expired.

If the value of the *bOverwrite* parameter is `true` and that URL is already in the cache, the file will be overwritten.

The user sets the cache size limit in the Central user preferences. The default size for space shared by all applications running in Central is 20 MB. All applications share this limited space. When the cache contents exceeds 20 MB, the user is asked for more space for local Internet files. If refused, the file is not cached. You can check for success by calling `inLocalInternetCache()`.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

http://www.mydomain.com/pub/myFile.swf  
http://applications.mydomain.com/pub/myFile.swf

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

**Note:** The size limitation for a URL to add to cache is 129 characters (URLs with more than 129 characters will not be added to cache).

## Shell.editLocationDialog()

### Availability

Macromedia Central.

### Usage

```
shellReference.editLocationDialog()
```

### Example

```
// this example gives the user a button they can use to launch the edit dialog
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // set up the button
    edit_btn.onPress=function()
    {
        gShell.editLocationDialog();
    }
}
```

### Parameters

None.

### Returns

Nothing.

### Description

Shell or Console method; called by your application or pod to open the Edit Location dialog box from the user's preferences, so that they may edit their current list of locations. This action is the same as if the user selects Edit Locations from the Location pop-up menu in the Identity & Location section of the general preferences. The only difference here is that the user never sees the rest of their preferences if they cancel the operation. Using the `editLocationDialog()` method is simply a way to help access this setting by way of your application.

## Shell.getAgent()

### Availability

Macromedia Central.

### Usage

```
agentData=shellReference.getAgent()
```

### Example

```
// this example lets the user start an agent if it's not already started
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // set up a button to start agent
    turnOnAgent_btn.onPress=function()
    {
        // use getAgent() to find the started property
        if(gShell.getAgent().started==true)
        {
            prompt_txt.text="Already running";
        }
        else
        {
            // attempt to start agent and report the results
            var result=gShell.startAgent();
            prompt_txt.text="Result: "+(result==true?"success":"failure");
        }
    }
};
```

### Parameters

None.

### Returns

*agentData*    Object; contains the following list of properties.

Element	Description
id	A unique numeric ID for the agent. This is the same value received as the second parameter when Central calls the <code>Agent.onActivate()</code> event handler.
name	A string that specifies the name of the agent, which is the same as the name declared for this agent in the <code>product.xml</code> file.

Element	Description
src	A string that specifies the fully qualified location of the SWF file implementing the agent, which is the same as the location declared for this agent in the product.xml file.
started	A Boolean value that indicates whether this agent has been started (that is, whether it's currently running).

Console or Shell method; called by your pod or application to ascertain various properties of the agent. Calling `getAgent()` returns an object with several properties. Considering that most of these properties are hard-wired in your product.xml file, the most useful properties are `started` and `enabled`.

## Shell.getBounds()

### Availability

Macromedia Central.

### Usage

```
onActivate=function(shell)
{
    // set a variable to reference the Shell
    gShell=shell;

    // trigger our own onResize to set the initial layout
    this.onResize();
}

// define the onResize callback
onResize = function()
{
    // ask shell to check app's min & max size
    var bounds = gShell.getBounds();

    // if the bounds weren't returned, use the stage size
    if (bounds == null)
    {
        bounds = {width: Stage.width, height: Stage.height};
    }
    // layout application items
    centered_mc._x = bounds.width / 2;
    centered_mc._y = bounds.height / 2;
};
```

### Parameters

None.

## Returns

An ActionScript object with two properties: `width` and `height`. These match the new size of your application.

*width* An integer that is the width of your application boundary.

*height* An integer that is the height of your application boundary.

## Description

Shell method; called by the application to request the current application size. This is used rather than using `Stage.width` and `Stage.height` directly, since there are nonapplication items (such as the toolbar and prompt area) that take up stage space.

You should always call `getBounds()` to determine the application size and layout elements to fill the bounds exactly. In conjunction with the `onResize()` event, you can make an application that automatically changes the layout as the user resizes Central. Your stage needs to react to `onResize()`, as this gets triggered when the user changes the window size or the zoom level in the Window menu.

## Shell.getNotices()

### Availability

Macromedia Central.

### Usage

```
arrayOfStructures=shellReference.getNotices()
```

### Example

```
// This example creates notice when the user starts or stops your app.
// It uses getNotices() so that it can remove any matching notices.
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    makeNonDuplicatedNotice("STARTUP");
};

onDeactivate=function()
{
    makeNonDuplicatedNotice("SHUTDOWN");
};

makeNonDuplicatedNotice=function(theType){
    // first see if there are any existing notices that match this type
    var currentNotices=gShell.getNotices();
    for(var i=0;i<currentNotices.length;i++)
    {
        var thisNotice=currentNotices[i];
        if(thisNotice.appData==theType)
        {
            gShell.removeNotice(thisNotice.id);
            break;
        }
    }
}
```

```

    }
}

// make a new notice
var now=new Date();
var noticeData = new Object();
var initialData = theType;

// make part unique
if(theType=="STARTUP")
{
    noticeData.name = "Start up time";
    noticeData.description = "You started this app at "+now.toString();
}
else
{
    noticeData.name = "Shut down time";
    noticeData.description = "You closed this app at "+now.toString();
}

// set the rest of the properties
noticeData.alert = false;
noticeData.navigate = false;
noticeData.engageString = null;
noticeData.timeout = 0;

gShell.addNotice(noticeData, initialData);
};

```

## Parameters

None.

## Returns

An array of structures, each with the following properties:

Property	Description
creationTime	Date object containing the exact time the notice was created.
appId	A unique numeric ID for the application that created the notice. This is the same value received by the <code>onActivate()</code> event.
id	A unique numeric ID for this notice. This is the same value returned when you call <code>addNotice()</code> .
initialData	Can be any data type, passed as the second parameter when you issue <code>addNotice()</code> . For details on how this lets you pack a notice with custom data, see <code>addNotice()</code> .
noticeData	An object that contains general information about the notice, as described next.

*noticeData* An object containing several properties with detailed information about the notice. The following are the available properties:

Property	Description
<code>id</code>	A number that represents the notice ID. This is the same ID returned when the notice is first created using <code>addNotice()</code> . That is, you don't set this property when you create the notice.
<code>name</code>	A string to be displayed in the notice's title bar. If not specified, the default value of <code>name</code> is: <i>application name</i> Notice
<code>description</code>	Longer string to be displayed in the notice's body. The default is an empty string.
<code>timeout</code>	A number that specifies the seconds after which the notice should be automatically dismissed. Set <code>timeout</code> to 0 to create a notice that never times out.
<code>alert</code>	A Boolean value that indicates whether the notice should be brought to the user's attention, rather than recorded in the Console.
<code>engageString</code>	A short string; displayed on the engage button. If not specified, no engage button appears.
<code>navigate</code>	A Boolean value that indicates whether the shell should start the appropriate application when the user selects engage.
<code>unread</code>	A Boolean value that indicates whether the notice has been viewed by the user.

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to get the currently active notices that your application created. The `getNotices()` method is one of many methods that help you manage the notices you produce. You don't want to inundate your users with useless notices.

When you invoke the `addNotice()` method, an ID number is returned (that you can use when call `removeNotice()`). When a notice is dismissed, the `onNoticeEvent()` callback triggers with complete details. Finally, you can always find complete details regarding the existing notices any time by using the `getNotices()` method.

## Shell.getPods()

### Availability

Macromedia Central.

### Usage

```
podData=shellReference.getPods()
```



## Example

```
// displays a list of all available pods and adds an option to remove
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
    refreshListofAllPods();

    myButton.onPress=function(){
        gShell.removePod(myListComponent.getSelectedItem().data);
        refreshListofAllPods();
    };
};
refreshListofAllPods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of all the pods
    var allPods=gShell.getPods();

    // loop through the pods extracting their names and IDs
    for (var i=0; i<allPods.length; i++)
    {
        var thisPod=allPods[i];
        // add this pod's name and id to an MListBox component instance
        myListComponent.addItem(thisPod.name, thisPod.id);
    }
};
```

For additional examples of this method, see [AgentManager.removePod\(\)](#).

## Parameters

None.

## Returns

An array of `podData` structures for all pods available to this application. The values for a `podData` structure are set in the `product.xml` file or by a script calling `addPod()` or, in the case of `appId` and `id`, by Central itself. For reference, the entire `podData` object documentation is shown below.

*podData* Pod data object containing several properties. This is the single parameter used any time you create, destroy, or simply reference a pod. A `podData` object includes the following properties:

Property	Description
<code>id</code>	A numeric unique ID returned when you call <code>addPod()</code> , so that you can later reference the pod. Do not set this property when adding the pod.
<code>name</code>	A string that specifies the name displayed on the pod itself. The name in the pop-up menu at the top of the Console always matches your application name. One application can have multiple pods, each with its own name.

Property	Description
<code>className</code>	<p>A string that specifies the name of the class, as defined in the product.xml file, that refers to a particular implementation of the pod. If your application uses only one pod, you won't need a <code>pod className</code>. However, if you plan to have multiple pod instances based on the same template, you should define both a <code>podClass</code> and your <code>pod</code>. You need to define the <code>podClass</code> element separately. For example, suppose that you describe the <code>podClass</code> tag as follows:</p> <pre>&lt;podClass name="className" src="pod.swf"/&gt;</pre> <p>You can then create instances of this <code>podClass</code> in one of two ways. First, using the product.xml file, you can add the following pod tag:</p> <pre>&lt;pod name="display name" className="className"/&gt;</pre> <p>The second way to create an instance of this <code>podClass</code> is when creating a new pod using <code>addPod</code>, as follows:</p> <pre>podData.name="display name"; podData.className="className";</pre> <p>Note: When using ActionScript 2.0, avoid <code>.class</code>; it is a reserved word. Use <code>className</code>.</p>
<code>height</code>	An optional numeric parameter that you may declare to set the height, in pixels, of the pod instance. The default height is 100. Pod widths are fixed at 170 pixels.
<code>src</code>	A string that specifies the source SWF file. The value is either absolute or relative. (You can only set this value in the product.xml file's <code>pod</code> tag or <code>podClass</code> tag.)
<code>enabled</code>	A Boolean value that specifies whether the pod has been added and is available to the user. This value is only returned when calling <code>getPods()</code> ; don't set it in the object you pass to <code>addPod()</code> .
<code>appId</code>	A number set by Central to associate a pod with your application.
<code>supportedTypes</code>	An array containing strings that identify the data types that this pod can exchange through the Blast feature. (You can only set these types in the product.xml file's <code>supportedTypes</code> tag within the <code>pod</code> or <code>podClass</code> tags.)
<code>initialData</code>	An optional property of any data type that you set at the time you trigger <code>addPod()</code> . You can determine this value later when you reference a pod.

AgentManager, Console, or Shell method; gets the list of all pods available to this application. This includes pods listed in the product.xml file as well as any created by the `addPod()` method. To get a list of only those pods currently arranged in the Console, use the `getViewedPods()` method instead.

Realize that your application will only be listed once in the Console's pod selection pop-up menu (provided your application has at least one pod). For an application with more than one pod, the user will see a secondary pop-up menu inside the pod (adjacent to the pod's name). The `getPods()` method returns an array of all the pods that will appear in that secondary pop-up menu.

For more information see `getViewedPods()`, `addPod()`, and `viewPod()`.

## Shell.getPreferences()

### Availability

Macromedia Central.

### Usage

```
prefObject=ref.getPreferences()
```

### Example

```
// Displays as customized a message as possible at startup
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    // get all the preferences
    var all=gShell.getPreferences();

    // prepare a field to populate
    message_txt.text="";

    // encourage them to enable background tasks, just in case
    if(all.agentsEnabled==false)
    {
        message_txt.text+="Please enable background tasks."+newline;
    }

    // if we can't find first or last name, use a generic message
    if(all.userData.firstName==null || all.userData.lastName==null)
    {
        message_txt.text+="Welcome!"+newline;
    }
    else
    {
        message_txt.text+="Welcome "+all.userData.firstName+" "
            +all.userData.lastName+"."+newline;
    }

    // if the locations value isn't null
    if(all.locations!=null)
    {
        // store the location profile from the appropriate index
        var here=all.locations[all.currentLocationIndex];

        // fashion a personalized message to display
        message_txt.text+="You're probably glad to be "
            +here.label+ " in beautiful "+here.city+".";
    }
};
```

## Returns

`pref0Object` Object containing details from the user's global preference settings. Depending on how much access the user has given to your application, you can find the values for some or all of the following properties.

Element	Description
<i>userData</i>	A structure with three properties: {firstName: xxx, lastName: xxx, email: xxx}
<i>locations</i>	An array of structures, each with the following properties: {label: xxx, address1: xxx, address2: xxx, city: xxx, state: xxx, zipcode: xxx, phone: xxx, country: xxx, latitude: xxx, longitude: xxx,}
<i>currentLocationIndex</i>	An index indicating the currently selected location (within the <code>locations</code> array).
<i>agentsEnabled</i>	A Boolean value that indicates whether agents are enabled.

## Description

AgentManager, Console, or Shell method; called by the pod or application to get the general Central preferences the user has exposed to your application. The value for the `agentsEnabled` property matches the user's setting for whether background tasks are allowed (set in the Advanced Preferences dialog box). This value is always available. In fact, you'll also see values for the `userData`, `locations`, and `currentLocationIndex` properties (based on settings under the Identity & Location Preferences dialog box). However, the values are all `null` by default and won't be available until the user has specifically allowed your application access to this data. It's easiest to visualize these properties and subproperties while viewing the Identity & Location Preferences dialog box.

## Shell.getSelectedItem()

### Availability

Macromedia Central.

### Usage

```
mySelectedItem=shellReference.getSelectedItem()
```

### Example

```
// In order for the Blast menu to appear, you need  
// at least one other application installed that accepts these data types.  
// To make testing easy, just use the following code in the product.xml file
```

```
//
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
  <type>any</type>
</supportedTypes>

// this example lets the user see the most recent data they Blasted
onActivate=function(shell)
{
  // set a variable to reference the Shell
  gShell=shell;

  // set up the SelectedItem instances
  var tint = new SelectedItem("http://www.mysite.com/schemas#", "aColor");
  tint.name= "Red";
  tint.value=0xFF0000;

  var shape = new SelectedItem("http://www.mysite.com/schemas#", "aShape");
  shape.name= "Square";
  shape.sideCount=4;

  // set button labels and store a reference to its SelectedItem object
  sendTint_btn.label = tint.name;
  sendTint_btn.data=tint;
  sendTint_btn.onRelease = pickItem;

  sendShape_btn.label = shape.name;
  sendShape_btn.data=shape;
  sendShape_btn.onRelease = pickItem;

  // create a button just to see it work
  review_btn.onRelease = function()
  {
    // grab the last selected item (in the first slot of the array)
    var lastItem=gShell.getSelectedItem()[0];
    message_txt.text="Last item selected was a " +
      lastItem.type + " named " + lastItem.name;
  };
};

// called by either MPushButton to display a message on screen
function pickItem(_pb)
{
  var theItem=_pb.data;
  gShell.setSelectedItem([item], "send "+item.name);
}
```

## Parameters

None.

## Returns

A SelectedItem object with the data from your current selection.

## Description

Shell method; called by your application to ascertain the most recent `SelectedItem` you created using `setSelectedItem()`. You first create a `SelectedItem` instance of a certain data type, populate it, and then call the `setSelectedItem()` method. From that point, the user sees the Blast menu (provided at least one other installed application supports the contained data type). Additionally, `setSelectedItem()` immediately sends the data using the Blast feature if the user has enabled the Auto Blast option.

The `getSelectedItem()` method simply returns the most recent `SelectedItem` that you created. A `SelectedItem` object isn't generated automatically when the user simply selects something (for instance, part of a text field). Rather, you not only need to define the data type being generated (and any you intend the application to receive), you also have to deliberately create and populate the `SelectedItem` instance.

For another Blast example, see [“Application.onSelectedItem\(\)” on page 179](#) and [“Shell.setSelectedItem\(\)” on page 372](#).

For more information on the Blast feature, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Shell.getViewedApplications()

### Availability

Macromedia Central.

### Usage

```
arrayOfApplicationRecs=shellReference.getViewedApplications()
```

### Example

```
// this example stops users from launching multiple instances of your app
function onActivate(shell)
{
    gShell=shell;

    var activeApps = gShell.getViewedApplications();

    if(activeApps.length>1)
    {
        myAlertDialog("You're already running this app in another window");
    }
};
```

### Parameters

None.

## Returns

An array of `applicationRecs` structures that contain details about each instance of your application. Each `applicationRec` structure has the following two properties:

Property	Description
<code>appId</code>	Number indicating the unique ID for your application. (All instances of your application share this number.)
<code>shellID</code>	Number indicating the unique ID for the particular shell running the application. (You can think of this as an ID for the Central shell.)

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of all the shell instances running your application. Naturally, this method only returns information about your applications. You can use the `getViewedApplications()` method to prevent users from launching multiple instances of your application (as the example shows). Also, if you develop a multi-window application, you can use the IDs gathered to set up unique LocalConnection channels (although it's often simpler to use the [LCService object](#)).

## Shell.getViewedPods()

### Availability

Macromedia Central.

### Usage

```
arrayOfPodStructures=shellReference.getViewedPods()
```

### Example

```
// displays a detailed list of currently viewed pods
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our refresh function
    refreshListOfActivePods();

    // give a button the ability to trigger our refresh function
    myButton.onPress=function()
    {
        refreshListOfActivePods();
    };
};
refreshListOfActivePods=function()
{
    // clear the list component to be populated
    myListComponent.removeAll();

    // get an array of the currently viewed pods
```

```

var activePods=gShell.getViewedPods();

// loop through the pods extracting some data for each one
for (var i=0; i<activePods.length; i++)
{
    var thisPod=activePods[i];
    var thisLabel=thisPod.podData.name+
        " (slot: "+thisPod.position+") "+
        ((thisPod.collapsed)?"is not open":"is open");

    myListComponent.addItem(thisLabel);
}
};

```

## Parameters

None.

## Returns

An array of structures for each pod currently visible. The structures have the following properties:

Property	Description
<code>viewerID</code>	Number indicating a unique ID for the pod instance. This is the same number received as the third parameter in the pod's <code>onActivate()</code> handler.
<code>position</code>	Number indicating the current ordinal position of the pod (not the pixel location). Counting from the top, the uppermost pod is in <code>position 0</code> , then <code>position 1</code> , and so on.
<code>collapsed</code>	A Boolean value that indicates whether the pod is in the collapsed state.
<code>podData</code>	A Pod data object as specified in the <code>product.xml</code> file or defined when you call <code>addPod()</code> . For details on the properties contained in a <code>podData</code> object, see <code>getPods()</code> .

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to get the list of this application's currently *viewed* pods in the console. These are simply pods positioned in the Console (regardless of whether the console happens to be open). To get a list of all initialized pods initialized (that is, pods available to the user), regardless of whether they've been loaded in the Console, see [AgentManager.getPods\(\) on page 154](#).

To determine whether the console is open, use [AgentManager.isConsoleOpen\(\)](#).

The `getViewedPods()` method only returns pods for your application.

While you can use the `removePod()` method (given the `id` property inside the `podData` property) this won't just close the pod but will actually remove it from the pods available to the user. There is no "close pod" method.



## Shell.inLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.inLocalInternetCache(url);
```

### Example

```
// this example function loads images with the ultimate user control
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // trigger our homemade function
    loadImage("background.jpg");
}

loadImage=function(theImage, override)
{
    var imagePath="http://www.mysite.com/images/"+theImage;

    // if the image is already present (or they're overriding)
    if(gShell.inLocalInternetCache(imagePath)==true || override==true)
    {
        //add the image and load it into a clip instance
        error_txt.text="Loading "+theImage;
        gShell.addToLocalInternetCache(imagePath);
        clipInstance.loadMovie(imagePath);
    }
    else
    {
        // if they're connected
        if(gShell.isConnected()==true)
        {
            // get their approval by making the button set override to true
            error_txt.text="Do you want to download "+theImage+"?";
            okay_btn.onPress=function(){ loadImage(theImage,true) };
        }
        else
        {
            // if they're not connected just let them try again
            error_txt.text="Connect then press the okay button";
            okay_btn.onPress=function(){ loadImage(theImage)};
        }
    }
};
```

## Parameters

*url* A string; fully qualified path that provides the location of the file to be added to the local Internet cache.

## Returns

A Boolean value; `true` if the URL is in the local Internet cache, `false` if not found.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to check if a URL is in the local Internet cache. There are several strategies that might require this method. Before requiring a user to endure a long download, you can first check if the file is available locally, in which case, you get the user's approval first. Also, if the user has indicated that they're not online and the file is not available locally, you can tell them they need to go online first. Finally, the `inLocalInternetCache()` method provides an indirect way to confirm that attempts to add files to the local Internet cache are successful, as described in the following best practice.

**Note:** Central considers file types of Portable Executable formats (DLL, EXE, OCX, and so on) unsafe and will not add them to the local Internet cache.

It's a good idea to always confirm the success of any `addToLocalInternetCache()` call by immediately issuing the `inLocalInternetCache()` method with an appropriate follow-up action if the method returns a value of `false`. This is because if the user's 20 MB cache is exceeded and they don't allow an increase, then `addToLocalInternetCache()` effectively fails.

When caching files, the files are identified by their URL. However, Central does not distinguish between separate hosts within the same domain. For example, Central considers the following two URLs as the same:

```
http://www.mydomain.com/pub/myFile.swf
http://applications.mydomain.com/pub/myFile.swf
```

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

## Shell.isConnected()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.isConnected()
```

### Example

```
// this example checks the connection state at startup
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
```

```

    // take current state and trigger onNetworkChange (where our code resides)
    this.onNetworkChange(gShell.isConnected());
};

onNetworkChange=function(connected)
{
    // save the connection state in a variable
    gOnline=connected;

    // display a visual online indicator
    onlineGraphic_mc._visible=gOnline;

};

```

### Parameters

None.

### Returns

A Boolean value: `true` if the user is connected, `false` if offline.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application to determine current network status. You should consolidate all your code related to connectivity inside the `onNetworkChange()` handler. Although Central triggers `onNetworkChange()` automatically, it only does so when the connection status *changes*. Therefore, you need to use `isConnected()` initially to bring your application in sync. The example shows how an application can trigger its own `onNetworkChange()` handler (though usually Central does this). This way, all the code is consolidated in one place. There's no reason to repeatedly check `isConnected()` from multiple places in your code.

## Shell.isConsoleOpen()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.isConsoleOpen()
```

### Example

```

// this example adds a notice in order to open a closed Console at startup
onActivate=function(shell)
{
    // set a variable to reference the AgentManager or Shell
    gShell=shell;

    if(gShell.isConsoleOpen()==false)
    {
        var noticeData = new Object();
        noticeData.alert = true;
    }
}

```

```

        noticeData.name = "Welcome to my app!";
        noticeData.description = "You'll need the Console in this app";
        gShell.addNotice(noticeData);
    }
};

```

#### Parameters

None.

#### Returns

A Boolean value: `true` if the Console is currently open, `false` if it is closed.

#### Description

AgentManager or Shell method; called by an agent or an application to determine if the Console is open. You might want to check whether the Console is open before you add or open new pods. Additionally, because some commands cause the Console to open (for example, adding a notice with its `alert` property set to `true`) you should first check whether the Console is open before deciding your approach.

Unlike most methods available from agents, applications, and pods, the `isConsoleOpen()` method is not available from a pod.

## Shell.isPurchased()

#### Availability

Reserved.

#### Description

Reserved. Interface currently unavailable. Applications that implemented previous versions of this method will continue to function: a call to `isPurchased` will return `true` for products previously purchased.

## Shell.newLocationDialog()

#### Availability

Macromedia Central.

#### Usage

```
shellReference.newLocationDialog([reqFields])
```

#### Example

```

// this prompts the user to create a new location with city and state required
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;

    newLocation_btn.onPress=function()

```

```

    {
        gShell.newLocationDialog(["locCity", "locState"]);
    }
};

```

## Parameters

*reqFields* An array that contains strings for the fields that you want to designate as required. The following table lists valid field names. It's easiest to visualize these field names while viewing the Identity & Location Preferences dialog box. (You can pass the literal string "noDialog" to open the Identity & Location Preferences dialog box without creating a new location entry or designating any required fields.)

Field	Description
"firstName"	User's first name
"lastName"	User's last name
"email"	User's e-mail address
"locAddress1"	The first line of the user's address
"locAddress2"	The second line of the user's address
"locCity"	User's city
"locState"	User's state
"locZip"	User's zip code
"locPhone"	User's phone number
"locLat"	User's latitude
"locLong"	User's longitude

## Returns

Nothing.

## Description

Shell or Console method; called by your application or pod to open the New Location dialog box from the user's preferences, so that they may create a new location and then edit the values. This is the same as if the user selects New Location from the Location pop-up menu in the Identity & Location section of the general preferences. The only difference here is that the user never sees the rest of their preferences if they cancel the operation.

To make no fields required, don't pass anything. To make the Identity & Location Preferences dialog box appear without creating a new location entry or designating any required fields, pass the literal string "noDialog".

Additionally, the `newLocationDialog()` method lets you designate that any or all fields are required. The user sees a small page curl on the required fields, and red lines around any fields they attempt to leave blank. (Be sure to pass an array containing *strings* that match the values listed in the table.)

## Shell.removeFromLocalInternetCache()

### Availability

Macromedia Central.

### Usage

```
shellReference.removeFromLocalInternetCache(URL)
```

### Example

```
// this example attempts to free up space in the user's cache when appropriate
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // add a new image to the local cache and load it into a clip
    var imagePath="http://www.mysite.com/images/";
    var theImage="photo.jpg";
    gShell.addToLocalInternetCache(imagePath+theImage);
    clipInstance.loadMovie(imagePath+theImage);

    // if the new image isn't present that means the cache is full
    if(gShell.inLocalInternetCache(imagePath+theImage)==false)
    {
        // take a list of previously loaded images (could be dynamic)
        myImageList=["big1.jpg", "big2.jpg", "big3.jpg"];

        // and remove each one
        for(var i=0;i<myImageList.length;i++)
        {
            gShell.removeFromLocalInternetCache(imagePath+myImageList[i]);
        }
    }
};
```

### Parameters

*url* Fully qualified location of the file to be removed from the local Internet cache.

### Returns

Nothing.

### Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a URL from the local Internet cache. Subsequent requests for that URL by any application in Central will retrieve that data from the web rather than from the cache.

The file in question must be *in* the local Internet cache for this method to work. That means that previously an application must have issued `addToLocalInternetCache()`. Another way to remove a file from the local Internet cache is by setting an expiration date when invoking `addToLocalInternetCache()`. Finally, the `Shell.addToLocalInternetCache()` method also has an `overwrite` parameter which effectively removes a file by replacing it. For more information, see `addToLocalInternetCache()`.

## Shell.removeNotice()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.removeNotice(noticeID)
```

### Example

```
// this example creates 3 notices and removes all 3 when any one is dismissed
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // create an array to store IDs for the notices we create
    gPostedNotes=new Array();
    //create a notice, set its name, and add it
    var noticeData = new Object();
    noticeData.engageString="Remove";
    noticeData.name = "One";
    gPostedNotes.push( gShell.addNotice(noticeData) );
    noticeData.name = "Two";
    gPostedNotes.push( gShell.addNotice(noticeData) )
    noticeData.name = "Three";
    gPostedNotes.push( gShell.addNotice(noticeData) )
};

onNoticeEvent=function(event, noticeData, initialData)
{
    // while gPostedNotes still has items remaining
    while(gPostedNotes.length>0)
    {
        // pop one off and remove it
        var thisNotice=gPostedNotes.pop();
        gShell.removeNotice(thisNotice);
    }
};
```

## Parameters

*noticeID* A number identifying the specific notice you want removed. You can use the number returned when you create a notice using the `addNotice()` method. You can also use an `id` property of any object within the array of notices objects returned from the `getNotices()` method.

## Returns

A Boolean value: `true` if the notice was removed, otherwise `false`.

## Description

AgentManager, Console, or Shell method; called by an agent, pod, or application, respectively, to remove a notice. You can only remove notices that your application created.

The most direct way to track IDs is to store them as you use `addNotices()`. However, this can be difficult because users can dismiss notices (in which case you'll have to trap the `onNoticeEvent()` event) and they might leave the notices untouched when they quit Central (in which case you'll have to save a `LocalShared` object). Probably the easiest tracking method is to include initial data in the second parameter of your `addNotice()` call. Use `getNotices()` and then step through each item returned looking for a matching `appData` property.

A best practice is to minimize the total number of notices by first deleting old notices and then replacing them with new ones containing up-to-date information. For an example of this practice, see [Shell.getNotices\(\)](#).

## Shell.removePod()

### Availability

Macromedia Central.

### Usage

```
shellReference.removePod(id)
```

### Example

```
// product.xml excerpt from within the <application> tag:
<podclass name="myRegularPod" src="pod.swf"/>
<podclass name="mySpecialPod" src="specialpod.swf"/>

<pod name="myDefaultPod" className="myRegularPod" />

// this example temporarily exposes a special pod for the user to open
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;

    // give the user the opportunity to access the special pod class
    gCounter=0;
    my_btn.onPress=function()
    {
```



```

gCounter++;
if(gCounter==3)
{
    // add a new pod
    var podData=new Object();
    podData.name="special #3";
    podData.className="mySpecialPod";
    gSpecialPodID=gShell.addPod(podData);
}
if(gCounter==4)
{
    // remove the special pod
    gShell.removePod(gSpecialPodID);
}
}
};

```

### Parameters

*id* A number returned when calling the `addPod()` method to create a pod. You can also use an `id` property of any object within the array of objects returned to the `getPods()` method.

### Returns

Nothing.

### Description

AgentManager, Console, or Shell object method; called by an agent, pod, or application, respectively, to remove a pod. Although this method removes from view any pods currently arranged in the Console, it works differently from the way the user manually closes a pod. In the case of `removePod()`, you aren't removing an individual pod SWF instance, but rather removing a pod instance once associated with your application. Naturally, if a matching pod is present in the Console, it must be closed, but using `removePod()` means the user will no longer be able to add that pod instance manually. There is no "close pod" method.

For more information about pods and pod classes see the entry for [Console.addPod\(\)](#). For more information about the difference between application pods and currently viewed pods, see the entries for [Console.getPods\(\)](#) and [Console.getViewedPods\(\)](#), respectively.

Central automatically removes all pods associated with an application when a user uninstalls the application.

## Shell.requestPayment()

### Availability

Reserved.

## Description

Reserved. Interface currently unavailable. Applications that implemented previous versions of this method will continue to function as non-purchased applications: a call to `requestPayment` will cause a notice to be displayed to the user, informing them that the feature is unavailable at this time.

## Shell.requestSizeChange()

### Availability

Macromedia Central.

### Usage

```
shellReference.requestSizeChange( width, height )
```

### Parameters

*width* An integer; the width you'd like your application to be.

*height* An integer; the height you'd like your application to be.

### Returns

Nothing.

## Description

Shell method; called by your application to request a change to the size of the application window. The shell might not be able to size the window as requested, for example if the user's monitor resolution is less than your request requires. Also, Central imposes a minimum screen size of 500 x 300 pixels and a maximum size of 1600 x 1600 pixels.

Any window sizing (including `requestSizeChange()`) results in the `onResize()` event. A best practice is to call `Shell.getBounds()` to determine the actual size, and then lay out your application elements to fill the bounds appropriately.

## Shell.setProgress()

### Availability

Macromedia Central.

### Usage

```
shellReference.setProgress( percent )
```

### Example

```
onActivate=function(shell){  
    // set a variable to reference the Shell  
    gShell=shell;  
  
    // indentify an image to load  
    var theFile="http://www.mysite.com/images/photo.jpg"  
  
    // check to see if we need to get it from the Internet  
    if(!gShell.inLocalInternetCache(theFile))
```

```

{

    // tell the user what's happening
    gShell.setStatus("downloading image");
    gShell.addToLocalInternetCache(theFile);

    // set up the loader script (including setting the progress)
    holderClip.onEnterFrame=function()
    {

        // only begin if the file is large
        if(this.getBytesLoaded(>5)
        {
            var ratio=this.getBytesLoaded()/this.getBytesTotal();
            gShell.setProgress(ratio*100);
            if (ratio==1)
            {
                gShell.setProgress(0);
                gShell.setStatus("");
                this.onEnterFrame=null;
            }
        }
    }

    // begin loading
    holderClip.loadMovie(theFile);
};

```

## Parameters

*percent* A number that represents the percentage of progress. The *percent* parameter can have the following values:

Value	Description
-1	Shows indeterminate progress (barber pole).
1-100	Shows the percentage of progress used for normal progress bar.
0 or over 100	Hides the bar.

## Returns

Nothing.

## Description

Shell method; called by your application to display the built-in progress bar at the bottom of the shell. Pass a *percent* parameter of -1 to show indeterminate time, which displays a barber pole animation; pass any number between 1 and 100 to see that value as a percentage; pass 0 to hide the progress bar (for instance, when you're done with the progress bar). You also use the `setStatus()` method to display a text message in the status bar. This is useful for labeling the action that the progress bar is indicating.

## Shell.setSelectedItem()

### Availability

Macromedia Central.

### Usage

```
shellReference.setSelectedItem ( data [,description] )
```

### Example

```
// RECEIVER APPLICATION:

// place the following in the product.xml file's pod and/or application section
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
  <type>any</type>
</supportedTypes>

// this method will populate an MListBox component
onSelectedItem=function(data)
{
  // prepare a List component to populate
  myListComponent.removeAll();

  for(var i=0;i<data.length;i++)
  {
    // make sure we treat the data as a selectedItem (not XML)
    var thisItem=data[i].asSelectedItem();
    myListComponent.addItem(thisItem.name, thisItem.description);
  }
};

// SENDER APPLICATION:

// this part of the code shows how an application can prepare data to send
onActivate=function(shell){
  // set a variable to reference the Shell or Console
  gShell=shell;

  send_btn.onPress=function()
  {
    // prepare the data as an array of two selectedItem items
    var dataToSend=new Array();

    var item1=new SelectedItem("http://www.mysite.com/ns#", "aType");
    item1.name="name one";
    item1.description="this is the description for item 1";
    dataToSend.push(item1);

    var item2=new SelectedItem("http://www.mysite.com/ns#", "aType");
    item2.name="name two";
    item2.description="this is the description for item 2";
    dataToSend.push(item2);

    // with the items prepared, set the data array and a prompt
```

```
        gShell.setSelectedItem([item1, item2], "blast two items!");
    };
};
```

## Parameters

**data** An array of instances of the `SelectedItem` ActionScript object or an array of XML nodes. The array may contain many instances of the same data type (for instance, a list of restaurants) or many instances of different pieces of data (for instance, a restaurant, a movie, and a date). If any of the elements do not have a data type associated with them, they are silently ignored.

**description** A string that you want Central to display in the status bar; this string describes the selected data. This parameter is optional but highly recommended, and should be a concise description of the data, since the description is displayed in the status bar next to the Blast menu button.

## Returns

Nothing.

## Description

Shell method; called by your application to notify the shell that there is new data that the user has entered or selected (or otherwise been made available), which can be pushed to other applications using the Blast menu. If the user has set the Auto Blast option, the data is sent using the Blast feature immediately after the `setSelectedItem()` method is called.

The application can call `setSelectedItem()` repeatedly to update the selected item and its description, as needed.

Additionally, your application can call `setSelectedItem(null)` to clear the selected item.

After calling `setSelectedItem()`, Central populates the Blast menu with a list of applications that support receiving any of the data types in the selected item's array. Central also holds the data so that you can retrieve it later by calling `getSelectedItem()`, even if the original `SelectedItem` instances objects are deleted. (Don't confuse this with the `getSelectedItem()` method that is available to many components.)

Before you trigger `setSelectedItem()`, you always need to first format the data as `SelectedItem` instances. Also, after you call `setSelectedItem()`, that data merely becomes available to other applications, although it's automatically sent if the Auto Blast option is enabled. The other applications must support receiving the data type that you're selecting, by specifying it in the `supportedTypes` tag of their respective `product.xml` files. Finally, after the data is sent through the Blast feature to another application, that application needs to have an `onSelectedItem()` callback defined that handles the process of receiving the data. Inside that code, the other application presumably does something interesting with the data, perhaps making a graph of the numbers received.

For more information about `SelectedItem` objects, see [“SelectedItem object” on page 335](#).

For more information about the `onSelectedItem()` event, see [“Application.onSelectedItem\(\)” on page 179](#).

For more information about the Blast feature and the `supportedTypes` tag, see [Chapter 7, “Using the Blast Feature,”](#) on page 105.

## Shell.setSelectedItem()

### Availability

Macromedia Central.

### Usage

```
shellReference.setSelectedItem ( data [,description] )
```

### Example

```
// RECEIVER APPLICATION:

// place the following in the product.xml file's pod and/or application section
<supportedTypes namespace="http://www.w3.org/2001/XMLSchema">
  <type>any</type>
</supportedTypes>

// this method will populate an MListBox component
onSelectedItem=function(data)
{
  // prepare a List component to populate
  myListComponent.removeAll();

  for(var i=0;i<data.length;i++)
  {
    // make sure we treat the data as a selectedItem (not XML)
    var thisItem=data[i].asSelectedItem();
    myListComponent.addItem(thisItem.name, thisItem.description);
  }
};

// SENDER APPLICATION:

// this part of the code shows how an application can prepare data to send
onActivate=function(shell){
  // set a variable to reference the Shell or Console
  gShell=shell;

  send_btn.onPress=function()
  {
    // prepare the data as an array of two selectedItem items
    var dataToSend=new Array();

    var item1=new SelectedItem("http://www.mysite.com/ns#", "aType");
    item1.name="name one";
    item1.description="this is the description for item 1";
    dataToSend.push(item1);

    var item2=new SelectedItem("http://www.mysite.com/ns#", "aType");
    item2.name="name two";
```

```

        item2.description="this is the description for item 2";
        dataToSend.push(item2);

        // with the items prepared, set the data array and a prompt
        gShell.setSelectedItem([item1, item2], "blast two items!");
    };
};

```

## Parameters

**data** An array of instances of the `SelectedItem` ActionScript object or an array of XML nodes. The array may contain many instances of the same data type (for instance, a list of restaurants) or many instances of different pieces of data (for instance, a restaurant, a movie, and a date). If any of the elements do not have a data type associated with them, they are silently ignored.

**description** A string that you want Central to display in the status bar; this string describes the selected data. This parameter is optional but highly recommended, and should be a concise description of the data, since the description is displayed in the status bar next to the Blast menu button.

## Returns

Nothing.

## Description

Shell method; called by your application to notify the shell that there is new data that the user has entered or selected (or otherwise been made available), which can be pushed to other applications using the Blast menu. If the user has set the Auto Blast option, the data is sent using the Blast feature immediately after the `setSelectedItem()` method is called.

The application can call `setSelectedItem()` repeatedly to update the selected item and its description, as needed.

Additionally, your application can call `setSelectedItem(null)` to clear the selected item.

After calling `setSelectedItem()`, Central populates the Blast menu with a list of applications that support receiving any of the data types in the selected item's array. Central also holds the data so that you can retrieve it later by calling `getSelectedItem()`, even if the original `SelectedItem` instances objects are deleted. (Don't confuse this with the `getSelectedItem()` method that is available to many components.)

Before you trigger `setSelectedItem()`, you always need to first format the data as `SelectedItem` instances. Also, after you call `setSelectedItem()`, that data merely becomes available to other applications, although it's automatically sent if the Auto Blast option is enabled. The other applications must support receiving the data type that you're selecting, by specifying it in the `supportedTypes` tag of their respective `product.xml` files. Finally, after the data is sent through the Blast feature to another application, that application needs to have an `onSelectedItem()` callback defined that handles the process of receiving the data. Inside that code, the other application presumably does something interesting with the data, perhaps making a graph of the numbers received.

For more information about `SelectedItem` objects, see [“SelectedItem object” on page 335](#).

For more information about the `onSelectedItem()` event, see [“Application.onSelectedItem\(\)” on page 179](#).

For more information about the Blast feature and the `supportedTypes` tag, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

## Shell.setStatus()

### Availability

Macromedia Central.

### Usage

```
shellReference.setStatus( message )
```

### Example

```
// this example displays a progress bar while downloading a remote image
onActivate=function(shell){
    // set a variable to reference the Shell
    gShell=shell;

    // identify an image to load
    var theFile="http://www.mysite.com/images/photo.jpg"

    // check to see if we need to get it from the Internet
    if(!gShell.inLocalInternetCache(theFile))
    {
        // tell the user what's happening
        gShell.setStatus("downloading image");
        gShell.addToLocalInternetCache(theFile);

        // set up the loader script (including setting the progress)
        holderClip.onEnterFrame=function()
        {
            // only begin if the file is large
            if(this.getBytesLoaded()>5)
            {
                var ratio=this.getBytesLoaded()/this.getBytesTotal();
                gShell.setProgress(ratio*100);
                if (ratio==1)
                {
                    gShell.setProgress(0);
                    gShell.setStatus("");
                    this.onEnterFrame=null;
                }
            }
        }

        //begin loading
        holderClip.loadMovie(theFile);
    };
};
```



## Parameters

*message* A string that specifies the message that is displayed in the Central status bar.

## Returns

Nothing.

## Description

Shell method; called by your application to set the current message in the status area of the shell window. The status area is at the bottom left of the Central main window (but to the right of the progress bar, if displayed). Use the *shellReference.setStatus("")* method to clear the status area.

## Shell.startAgent()

### Availability

Macromedia Central.

### Usage

```
shellReference.startAgent()
```

### Example

```
// this example stops and starts the agent as connection status changes
onActivate=function(shell)
{
    // set a variable to reference the Shell or Console
    gShell=shell;
};

onNetworkChange=function(connected)
{
    if(connected==false)
    {
        gShell.stopAgent();
    }
    else
    {
        gShell.startAgent();
    }
};
```

### Parameters

None.

### Returns

A Boolean value: *true* if the agent was started, *false* if offline.

## Description

Console or Shell method; called by a pod or an application to start the agent associated with your application as defined in the product.xml file. The product.xml file also lets you set your agent to start automatically every time Central starts. Simply set the `started` attribute to `true` inside the agent tag, as the following code shows:

```
<agent name="myAgent" src="agent.swf" started="true"/>
```

Most often, you use the `startAgent()` method after you issue a `stopAgent()` call.

## Shell.stopAgent()

### Availability

Macromedia Central.

### Usage

```
myBoolean=shellReference.stopAgent()
```

### Example

```
// this example stops and starts the agent as connection status changes
onActivate=function(shell)
{
    // set a variable to reference the AgentManager, Shell, or Console
    gShell=shell;
};

onNetworkChange=function(connected)
{
    if(connected==false)
    {
        gShell.stopAgent();
    }
    else
    {
        // this only works from an application or pod (not agent)
        gShell.startAgent();
    }
};
```

### Parameters

None.

### Returns

A Boolean value: `true` if the agent was successfully stopped, otherwise `false`.

## Description

AgentManager, Console, or Shell method; called by an agent itself, a pod, or an application to stop the agent SWF file (listed in the product.xml file). After it stops, the agent won't start again until you call `startAgent()` from an application or pod instance. In fact, agents do not start automatically unless the product.xml file's `agent` tag includes the attribute `started="true"`.

**Note:** Your application can only have one agent.

A best practice is to keep as much code as possible in your agent. In such a case, the `stopAgent()` method has questionable value. However, if your agent is primarily executing background tasks using `setInterval()`, it might be easiest to first clear all the intervals and then simply call `stopAgent()`. Additionally, if you design an application to use the agent only temporarily, then it makes sense to use `stopAgent()` (and `startAgent()`).

**Note:** You don't need to remove agents when the user uninstalls your application; Central does this for you.

## Shell.validateActivationKey()

### Availability

Reserved.

### Description

Reserved. Interface currently unavailable. Applications that implemented previous versions of this method will continue to function: a call to `validateActivationKey` will validate activation keys for products previously purchased.

## Shell.viewPod()

### Availability

Macromedia Central.

### Usage

```
shellReference.viewPod( podID [, bForce] )
```

### Example

```
// This example adds a pod only after the user first runs the app.  
// It requires you to define a PodClass named "post_install" in the product.xml  
  
onActivate=function(shell)  
{  
    // set a variable to reference the AgentManager, Shell, or Console  
    gShell=shell;  
  
    // check to see if the pod is installed  
    var foundID=null;  
    var allPods=gShell.getPods();
```

```

for(var i=0;i<allPods.length;i++)
{
    if(allPods[i].className=="post_install")
    {
        foundID=allPods[i].id;
        gPodID=foundID;
        break;
    }
}

// create it if it wasn't found
if(foundID==null)
{
    var podData=new Object();
    podData.name="Post Install Pod";
    podData.className="post_install";
    gPodID=gShell.addPod( podData);
}

// give the user a way to view the pod
podOpen_btn.onPress=function()
{
    // should first use getViewedPods to avoid multiple views of this pod
    //as is, this will view the pod added or found above
    gShell.viewPod(gPodID);
};
};

```

### Parameters

*podID* The number returned when calling `addPod()` to create a pod. You can also use an `id` property of any object within the array of objects returned to the `getPods()` method.

*bForce* Optional parameter that forces the creation of a new pod viewer in the Console. The *bforce* parameter is a Boolean value. If *true* (or omitted), `viewPod()` forces the creation of a new pod viewer in the Console. If *false*, a new pod viewer is created only if the pod referred to by `podID` is not already viewed in the Console (so no duplicate pods appear).

### Returns

Nothing.

### Description

AgentManager, Console, or Shell method; called by an agent, application, or pod, respectively, to open a specific pod instance so that it becomes visible in the top slot of the Console. All of the other visible pods are moved down in the console.

To call `viewPod()` you need a `podID` parameter. This means that you either must have used a script to call `addPod()`, which returns an `id`, or used the `getPods()` method to get an array full of structures (each one including an `id`). Using the `getPods()` method might seem to be haphazard if your application has more than one pod, as you wouldn't know which item in the array was for which pod. However, the data in the array returned from `getPods()` includes other details, including any initial data that you can specify in the `product.xml` file. It is probably most intuitive to simply use `viewPod()` immediately after the creation of a pod by using the `addPod()` method, as the example shows.

A best practice is to only call the `viewPod()` method in response to a direct user action. The user should choose how to best populate the Console.

## SOAPCall object

**ActionScript Class Name**    `mx.central.services.SOAPCall`

The SOAPCall object is part of the `mx.central.services` package and is intended as an advanced feature to be used with the WebService object (see [“WebService object” on page 384](#)).

When you create a new WebService object, it contains the methods corresponding to operations in the WSDL URL you pass in. Behind the scenes, a SOAPCall object is created for each operation in the WSDL as well. The SOAPCall is the descriptor of the operation, and as such contains all the information about that particular operation (how the XML should look on the wire, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall for a given operation by using the `getCall(operationName)` function. There is a single SOAPCall for each operation, shared by all active calls to that operation. Once you have the SOAPCall, you can customize the descriptor, by doing the following:

- Turn on or off decoding of the XML response.
- Turn on or off the delay of converting SOAP arrays into ActionScript objects.
- Change the concurrency configuration for a given operation.
- Add a header to the SOAPCall object.

### Method summary for the SOAPCall object

Method	Description
None.	

### Property summary for the SOAPCall object

Property	Description
<code>SOAPCall.doDecoding</code>	Turns on or off decoding of the XML response.
<code>SOAPCall.doLazyDecoding</code>	Turns on or off the delay of turning SOAP arrays into ActionScript objects.

## Event handler summary for the SOAPCall object

---

Event handler	Description
None.	

---

## Constructor for the SOAPCall object

### Availability

Macromedia Central

### Description

The SOAPCall object is not constructed by the developer. Instead, when you call a method on a WebService object, the WebService object returns a PendingCall object. To access the associated SOAPCall object, use `myPendingCall.myCall`.

## SOAPCall.doDecoding

### Availability

Macromedia Central

### Usage

*SOAPCall.doDecoding*

### Description

Property; turns on or off decoding of the XML response. By default, the XML response is converted (decoded) into ActionScript objects. If you want just the XML, you can set `SOAPCall.doDecoding = false`.

## SOAPCall.doLazyDecoding

### Availability

Macromedia Central

### Usage

*SOAPCall.doLazyDecoding*

### Description

Property; turns on or off “lazy decoding” of arrays. By default, use a “lazy” decoding algorithm to delay turning SOAP arrays into ActionScript objects until the last moment—this makes operations return a lot faster when large data sets are returned. This means that any arrays you get back from the remote end are ArrayProxy objects. Then, when you access a particular index (`foo[5]`), that element is automatically decoded if necessary. This behavior can be turned off, which causes all arrays to be fully decoded, by setting `SOAPCall.doLazyDecoding = false`.

# String object

**ActionScript Class Name**   String

Central supports the `String.replace` method as a convenience wrapper around `RegExp.replace`. You provide as regular expressions the string to find and what it should be replaced with as the two parameters, `re` and `replacement`, respectively. For more information about regular expressions, see [“RegExp object” on page 325](#).

## Method summary for the String object

Method	Description
<code>String.replace()</code>	Convenience wrapper around <code>RegExp.replace</code> .

## Flag summary for the String object

Flag	Meaning	Description
g	global	Creates a new HTTP object with the given base URL.
i	ignoreCase	Performs a caseless match.
m	multiline	Interprets the caret character (^) and dollar sign (\$) relative to embedded new lines. (By default, they are relative to the beginning and end of the string.)
s	dotall	Indicates that dot (.) matches new lines. (By default, it doesn't match.)
x	extended	Ignores whitespace in the <i>re</i> parameter. Use \s instead (for readability).

## String.replace()

### Availability

Macromedia Central.

### Usage

`String.replace(re, replacement)`

### Parameters

*re*   String to replace, in the form of a regular expression.

*replacement*   String that should replace the found string specified by the *re* parameter.

For more information about regular expressions, see [“RegExp object” on page 325](#).

### Returns

String; value of replacement string. The original String is left untouched.

## Description

String method; convenience wrapper around `RegExp.replace`. You provide as regular expressions the string to find and what it should be replaced with as the two parameters, `re` and `replacement`, respectively. This method returns a `String`. The value of which has the replacement made. The original `String` is left untouched.

## WebService object

**ActionScript Class Name** `mx.central.services.WebService`

The `WebService` object is part of the `mx.services` package and is intended to be used with the following objects:

- [Log object](#)
- [PendingCall object](#)
- [SOAPCall object](#)

The `WebServices` object acts as a local reference to a remote web service. When you create a new `WebService` object, the WSDL file that defines the web service gets downloaded, parsed, and placed in the object. You can then call the methods of the web service directly on the `WebService` object, and handle any callbacks from the web service. When the WSDL has been successfully processed and the `WebService` object is ready, the `onLoad()` callback is invoked. If there is a problem loading the WSDL, the `onFault()` callback is invoked.

When you call a method on a `WebService` object, the return value is a callback object. The object type of the callback returned from all web service method invocations is `PendingCall`. These objects are normally not constructed by developers, but instead are constructed automatically as a result of the `webServiceObject.webServiceMethodName()` command. These objects are not the result of the `WebService` call, which comes later. Instead, the `PendingCall` object represents the call in progress. When the `WebService` operation completes (usually several seconds after a method call is made), the various `PendingCall` data fields are filled in, and the `onResult` or `onFault` callback you provide is called. For more information about the `PendingCall` object, see [“PendingCall object” on page 306](#).

The Player queues up any calls you make before the WSDL is parsed, and attempts to execute them after parsing the WSDL. This is because the WSDL contains information that is necessary to correctly encode and send a SOAP request. Function calls that you make after the WSDL has been parsed do not need to be queued; they happen immediately. If a queued call doesn't match the name of any of the operations defined in the WSDL, Flash Player returns a fault to the callback object you were given when you originally made the call.

**Flash MX 2004 ActionScript Class Name** `mx.central.services.WebService`

## Method summary for the WebService object

Method	Description
<code>WebService.myMethodName()</code>	Invokes a specific web service operation defined by the WSDL.
<code>WebService.getCall()</code>	Gets the <code>SOAPCall</code> for a given operation



## Property summary for the WebService object

Property	Description
None.	

## Event handler summary for the WebService object

Event handler	Description
<a href="#">WebService.onFault()</a>	Called when an error occurred during WSDL parsing.
<a href="#">WebService.onLoad()</a>	Called when the web service has successfully loaded and parsed its WSDL file.

## Using the WebServices API

The WebServices API, included under the `mx.central.services` package, consists of the `WebService` object, the `Log` object, and the `PendingCall` object.

## Supported types

The `WebService` feature supports a subset of XML schema types, as defined in the following tables.

Complex types and the SOAP-Encoded Array type are also supported, and these may be composed of other complex types, arrays, or built-in XML schema types.

### Numeric simple types

XML schema type	ActionScript binding
<code>decimal</code>	Number
<code>integer</code>	Number
<code>negativeInteger</code>	Number
<code>nonNegativeInteger</code>	Number
<code>positiveInteger</code>	Number
<code>long</code>	Number
<code>int</code>	Number
<code>short</code>	Number
<code>byte</code>	Number
<code>unsignedLong</code>	Number
<code>unsignedShort</code>	Number
<code>unsignedInt</code>	Number
<code>unsignedByte</code>	Number

XML schema type	ActionScript binding
float	Number
double	Number

### Date and Time Simple types

XML schema type	ActionScript binding
date	Date object
datetime	Date object
duration	Date object
gDay	Date object
gMonth	Date object
gMonthDay	Date object
gYear	Date object
gYearMonth	Date object
time	Date object

### Name and String Simple types

XML schema type	ActionScript binding
string	ActionScript String
normalizedString	ActionScript String
QName	mx.services.Qname object

### Boolean type

XML schema type	ActionScript binding
Boolean	Boolean

### Object types

XML schema type	ActionScript binding
Any	XML object
Complex Type	ActionScript object composed of properties of any supported type
Array	ActionScript array composed of any supported object or type

### Supported XML schema object elements

```

schema
  complexType
    complexContent
      restriction
sequence | simpleContent
  restriction

```

## WebService security

The WebService API conforms to the Flash Player security model.

### User authentication and authorization

The authentication and authorization rules are the same for the WebService API as they are for any XML network operation from Flash. SOAP itself does not specify any means of authentication and authorization. For example, when the underlying HTTP transport returns an HTTP BASIC response in the HTTP Headers, the browser responds by presenting a dialog box for the user and subsequently attaching the user's input to the HTTP Headers in subsequent messages. This mechanism exists at a level lower than SOAP, and is part of the Flash HTTP authentication design.

### Message integrity

Message-level security involves the encryption of the SOAP messages themselves, at a conceptual layer above the network packets on which the SOAP messages are delivered.

### Transport security

The underlying network transport for Flash Player SOAP web services is always HTTP POST. Therefore, any means of security that can be applied at the Flash HTTP transport layer—such as SSL—is supported through web services invocations from Flash. SSL/HTTPS provides the most common form of transport security for SOAP messaging, and use of HTTP BASIC authentication, coupled with SSL at the transport layer, is the most common form of security for websites today.

## Constructor for the WebService object

### Availability

Macromedia Central

### Usage

```
myWebServiceObject = new WebService(wsdlURI [, logObject]);
```

### Parameters

*wsdlURI* URL of the web service WSDL file.

*logObject* Optional parameter that specifies the name of the Log object for this web service (see [“Log object” on page 300](#)).

### Returns

Nothing.

## Description

To create a `WebService` object, you call `new WebService()` and provide a WSDL URL. Flash Player returns a `WebService` object. The `WebService` object constructor can optionally accept a `Log` object and a proxy URL, as follows:

```
myWebServiceObject = new WebService(wsdlURI [, logObject]);
```

If you want, you can use two callbacks for the `WebService` object. Flash Player calls the `webServiceObject.onLoad(WSDLDocument)` function when it finishes parsing the WSDL file and the object is complete. This is a good place to put code that you want to execute only after the WSDL file has been completely parsed. For example, you might choose to put your first web service method call in this function.

Flash Player calls the `webServiceObject.onFault(fault)` function when an error occurs in finding or parsing the WSDL file. This is a good place to put debugging code and code that tells the user that the server is unavailable, that they should try again later, or similar information. For more information, see the individual entries for these functions.

## Invoking a web service operation

You invoke a web service operation as a method directly available on the web service. For example, if your web service has the method `getCompanyInfo(tickerSymbol)`, invoke the method in the following manner:

```
myPendingCallObject = myWebServiceObject.getCompanyInfo(tickerSymbol);
```

In this example, the callback object is named `myPendingCallObject`. All method invocations are asynchronous, and return a callback object of type `PendingCall`. (Asynchronous means that the results of the web service call are not available immediately.)

When you make the following call:

```
x = stockService.getQuote("macr");
```

the object `x` is not the result of `getQuote`; it's a `PendingCall` object. The actual results are only available later (usually several seconds later), when the web service operation completes. Your `ActionScript` code is notified by a call to the `onResult` callback function.

**Handling the `PendingCall` object** This callback object is a `PendingCall` object that you use for handling the results and errors from the web service method that was called (see [“PendingCall object” on page 306](#)). For example:

```
myPendingCallObject = myWebServiceObject.myMethodName(param1, ..., paramN);
myPendingCallObject.onResult = function(result)
{
    outputField_txt.text = result
};
myPendingCallObject.onFault = function(fault)
{
    debugField_txt.text = fault.faultCode + "," + fault.faultstring;

    // add code to handle any faults, for example, by telling the
    // user that the server isn't available or to contact technical
    // support
};
```

## WebService.getCall()

### Availability

Macromedia Central

### Usage

```
mySOAPCallObj=myWebServiceObject.getCall(operationName);
```

### Parameters

*operationName* The web service operation of the corresponding SOAPCall that you want to retrieve.

### Returns

A SOAPCall object.

### Description

WebService method; contains the methods corresponding to operations in the WSDL URL that you pass in, when you create a new WebService object. Behind the scenes, a SOAPCall object is created for each operation in the WSDL as well. The SOAPCall is the descriptor of the operation, and as such contains all the information about that particular operation (how the XML should look on the wire, the operation style, and so on). It also provides control over certain behaviors. You can get the SOAPCall for a given operation by using the `getCall(operationName)` method. There is a single SOAPCall for each operation, shared by all active calls to that operation. After you have the SOAPCall, you can change the operator descriptor by using the SOAPCall API. For more information, see [“SOAPCall object” on page 381](#).

### Example

For an example on using this call, see [“SOAPCall object” on page 381](#).

## WebService.onFault()

### Availability

Macromedia Central

### Usage

```
myWebServiceObject.onFault=function(fault)
{
    //handle the error
};
```

### Parameters

*fault* Decoded ActionScript object version of the error with properties. If the error information came from a server in the form of XML, the SOAPFault object is the decoded ActionScript version of that XML.

The type of error object returned to `webservice.onFault()` methods is a `SOAPFault` object. It is not constructed directly by developers, but is returned as the result of a failure. This object is an ActionScript mapping of the SOAP Fault XML type.

---

SOAPFault property	Description
<code>faultcode</code>	String; the short standard QName that describes the error.
<code>faultstring</code>	String; the human-readable description of the error.
<code>detail</code>	String; the application-specific information associated with the error, such as a stack trace or other information returned by the web service engine.
<code>element</code>	XML; the XML object representing the XML version of the fault.
<code>faultactor</code>	String; the source of the fault, optional if an intermediary is not involved.

---

## Returns

Nothing.

## Description

WebService callback function; Flash Player calls this function when the new `WebService(wsdlURI)` method has failed and returned an error. This can happen when the WSDL file cannot be parsed or the file cannot be found. The fault parameter is an ActionScript `SOAPFault` object.

## Example

The following example handles any error returned from the creation of the `WebService` object:

```
myWebServiceObject.onFault = function(fault)
{
    // captures the fault
    debugOutputField_txt.text = fault.faultstring;

    // add code to handle any faults, for example, by telling the
    // user that the server isn't available or to contact technical
    // support
};
```

## WebService.onLoad()

### Availability

Macromedia Central

### Usage

```
myWebServiceObject.onLoad=function(wsdlDocument)
{
    //execute startup code
};
```

### Parameters

*wsdlDocument*    WSDL XML document.

## Returns

Nothing.

## Description

WebService callback function; Flash Player calls this callback function when the WebService object has successfully loaded and parsed its WSDL file. Operations can be invoked in an application before this event occurs, but when this happens they are queued internally and not actually transmitted until the WSDL has loaded.

## Example

The following example specifies the WSDL URL, creates a new web service object, and receives the WSDL document after loading:

```
// specify the WSDL URL
var wsdlURI = "http://www.flash-db.com/services/ws/companyInfo.wsdl";

// creates a new web service object
stockService = new WebService(wsdlURI);

// receives the WSDL document after loading
stockService.onLoad = function(wsdlDocument)
{
    // code to execute when the WSDL loading is complete and the
    // object has been created
};
```

## WebService.myMethodName()

### Availability

Macromedia Central

### Usage

```
callbackObj = myWebServiceObject.myMethodName(param1, ... paramN);
```

### Parameters

Parameters required depend on the web service method being called.

### Returns

*callbackObj* A PendingCall object to which you can attach a function for handling results and errors on the invocation. For more information, see [“PendingCall object” on page 306](#).

The callback invoked when the response comes back from the WebService method is PendingCall.onResult() or PendingCall.onFault(). By uniquely identifying your callback objects, you can manage multiple onResult callbacks, as the following example shows:

```
myWebService = new WebService("http://www.myCompany.com/myService.wsdl");
callback1 = myWebService.getWeather("02451");
callback1.onResult = function(result)
{
    // do something
};
```

```
callback2 = myWebService.getDetailedWeather("02451");
callback2.onResult = function(result)
{
    // do something else
};
```

### Description

Web service method; to invoke a web service operation, invoke it as a method directly available on the web service. For example, if your web service has the method `getCompanyInfo(tickerSymbol)`, make the following call:

```
myCallbackObject.myservice.getCompanyInfo(tickerSymbol);
```

All invocations are asynchronous, and return a callback object, of the object type `PendingCall`.

## XML object

**ActionScript Class Name** XML

Central extends the Flash XML object to assist in setting data types for XML tags that you want to add to your chosen data schema. For more information see, [“Using XML objects to send data” on page 112](#).

### Method summary for the XML object

Method	Description
<a href="#">XML.setType()</a>	Sets the data type for a custom XML element.

### Property summary for the XML object

Property	Description
<code>None.</code>	

### Event handler summary for the XML object

Event handler	Description
<code>None.</code>	

## XML.setType()

### Availability

Macromedia Central.

### Usage

```
function setType(namespace:String, element_type:String):Void definition
data.setType("http://www.myapp.com/my_Element#", "my_Element");
```

### Parameters

*namespace* The `xmlns` attribute on the root node.



*element\_type* The `xsi:type` attribute of the element.

## Returns

Nothing.

## Description

XML method; sets the data type for a custom XML element. Use this method to inform Central of the data type of the XML element that you created with the call to `XML.createElement("source")`. For more information see, [“Using XML objects to send data” on page 112](#).



# CHAPTER 11

## The product.xml File

The product.xml file is the XML descriptor file for your product. The product.xml file describes your product, including all of the applications and their associated pods and agents. The product.xml file must also include a product ID obtained when you register your product with Macromedia

When you publish your application, post the product.xml file in the same domain as your product files. During the installation process, Macromedia Central confirms that the files listed in the product.xml file come from the same domain as the product.xml file itself. This helps ensure that the product is what it claims to be and that no malicious programmer is using a legitimate product.xml file to install illegitimate program files.

Macromedia recommends naming this file *product.xml* and placing it in the same directory as your main application SWF file.

### Sample product.xml file

The following is a simple example of a product.xml file:

```
<product
  name="myProduct"
  vendor="My Company"
  created="12/21/02"
  productID="XXXXX-XXXXX-XXXXX-XXXXX"
  version="1"
  price="Free"
  license="http://www.mycompany.com/Central/myApp/eula.htm"
  requireVersion="1.5"
  category="Other">

  <description>"Internet News Reader"</description>

  <author
    name="Joe Culayd"
    info="http://www.mycompany.com"/>

  <icon
    src="http://www.mycompany.com/Central/myApp/1st_icon.swf"
    size="40"/>
```

```

<application
  name="1st_of_many"
  src="http://www.mycompany.com/Central/myApp/1st_of_many.swf"
  version="1"
  help="http://www.mycompany.com/Central/myApp/app_help.html"
  info="http://www.mycompany.com/Central/myApp/app_info.html"
  background="#000000"
  lang="en"
  category="Other"
  sendto="true"
  hasPreferences="true">

  <description>
    "This is my story...."
  </description>

  <author
    name="Joe Culayd"
    info="http://www.mycompany.com"/>

  <icon
    src="http://www.mycompany.com/myIcons/app_icon.swf"
    size="40"/>

  <pod
    name="myPod"
    class=""
    src="http://www.mycompany.com/Central/myApp/myPod.swf"
    enabled="1"
    height="90">

  </pod>

  <agent
    name="myAgent"
    src="http://www.mycompany.com/Central/myApp/myAgent.swf"
    started="true">

  </agent>
</application>
</product>

```

**Note:** Your product.xml file can include either absolute URLs or partial (relative) URLs when specifying the various external files that your product comprises. Relative URLs resolve relative to the location of the product.xml file.

## Product.XML schema

In the following sections, the product.xml schema is described in the order it appears in a typical product.xml file. A product **tag** can include many application and file **tags**. An application tag can include multiple pod, icon, author, and file tags, but only one agent, initialData, and supportedTypes tag.

# product element

## Description

Root tag; provides all of the elements and attributes that describe your product.

## Usage

```
<product name="myApp"
  created="04/30/2003"
  version="1.5"
  price="Free"
  license="http://www.mycompany.com/Central/myApp/eula.htm"
  requireVersion="1"
  category="Personal Productivity"
  vendor="Central Product Builder"
  productID="XXXXX-XXXXX-XXXXX-XXXXX">
```

## Attributes

**name** Required attribute. Your product name.

**created** Optional attribute. The date that your product was created, in the form of mm/dd/yyyy. This attribute is required if you list your application in the Application Finder.

**version** Required attribute. Your product version.

**price** Required attribute. Your product price. If your product is free, set the price to 0.

**license** Optional attribute. You can specify a URL for your application's end user license agreement.

**requireVersion** Optional attribute. Required version of Central. Currently, this value should be set to 1.5 if you require the user to have Central 1.5 installed.

**category** Required attribute. The category that your product will be listed under in the Application Finder. Your application is published in the Application Finder under the values you selected in the Application Publisher, and not necessarily in the value you provided in the product.xml file. When you publish your application, the Macromedia Central Application Publisher reads the product.xml file and use the corresponding default value in the category list. You have the option of changing the value or adding more categories at this time. The following strings are possible values for the category attribute and must be matched exactly so that the Application Publisher can set the default category:

- Business & Professional
- Education
- Entertainment
- Games
- Health & Fitness
- Medical
- News & Weather
- Personal Productivity

- Reference
- Shopping
- Travel
- Utilities
- Other
- Our Picks (reserved string for Macromedia, Incorporated use only)

When you publish your application, if there is not an exact match, you must select at least one category or you will be unable to complete registration of your application.

`vendor` Optional attribute. Company name displayed in the Macromedia Central user interface. This attribute is required if the product is not free.

`productID` Required attribute. Unique identification number for your product, received when you registered your product with Macromedia.

## author element

### Description

Child element of the `Application` tag and the `Product` tag; string to describe the author of either the product or one of its applications.

### Usage

```
<author name="Esmeralda Ultima" info="http://authorHost.domain.com/" />
```

### Attributes

`name` Required attribute. Author's name that appears in the application's About dialog box and in the Central Application Finder.

`info` Optional attribute. Absolute URL of the website associated with the author's name.

# icon element

## Description

Child element of the `Application` tag and/or the `Product` tag; filename that points to the icon for either the product or one of its applications. An icon for an application should be a SWF file. The recommended size is 40 pixels. For more information about providing icons for your application, see [Chapter 1, “Getting Started,” on page 11](#).

## Usage

```
<icon
  src="/myProductIcon.swf"
  size="40"
/>
```

## Attributes

`src` Required attribute. Location of your product icon file. May be represented as an absolute URL, for example, `http://productHost.domain.com/products/myProduct/icons/myProductIcon.swf`, or as a pathname relative to the location of the `product.xml` file, for example, `icons/myProductIcon.swf`.

`size` Required attribute. Size of your icon in pixels, either the width or height dimension, whichever is larger.



# description element

## Description

Child element of the `product` and `application` tags. Describes the application. The Description string is displayed in the details view of the Application Finder. Only the first 128 characters are displayed.

The basic rules for escaping XML apply to this string, but Central might not decode XML entities. Avoid putting HTML code in your description.

## Usage

```
<description>"Pick-a-Product helps you..."</description>
```

## file element

### Description

Child element of the `Application` tag; filename that points to a supplemental file for the application.

When caching files, the files are identified by their URL. Central (unlike previous versions) can distinguish between separate hosts within the same domain. For example, Central *no longer* considers the following two URLs the same:

- `http://www.mydomain.com/pub/myFile.swf`
- `http://applications.mydomain.com/pub/myFile.swf`

You can cache files from multiple hosts within a domain as long as the paths to the files are unique across these hosts.

**Note:** The size limit of a file being cached is 20 MB. During your application's installation, if one or more of the files are over 20 MB, the over-limit file is not downloaded and the Central user receives an error indicating that some files were not downloaded.

### Usage

```
<file src="http://www.mycompany.com/directoryName/myStrings.txt"/>
```

### Attributes

`src` Required attribute. Location of a file.

Files specified with this tag are cached locally by Central when the application is installed. This allows faster access to the file and allows the file to be accessed when Central is in offline mode. The files being cached should be from the same domain as the application. To specify multiple files to be cached, include multiple `file` tags.

Files are stored in the directory on the user's computer named for the application's domain.

Each file should be represented as an absolute URL, for example, `http://productHost.domain.com/products/myProduct/myStrings/myStrings.txt`, or as a pathname relative to the location of the `product.xml` file, for example, `myStrings/myStrings.txt`.

# application element

## Description

Child element of the `product` tag; a Flash SWF application file associated with the product defined in the `product.xml` file. Each application residing with the product should have its own application tag.

## Usage

```
<application
  name="My Product"
  shortname="My Prod"
  src="http://productHost.domain.com/products/myProduct/myApp/myApp.swf"
  background="#CCCCFF"
  help="http://host.domain.com/products/myProduct/myApp/Help/appHelp.html"
  info="http://host.domain.com/products/myProduct/myApp/productInfo.html"
  lang="en"
  version="1"
  category=""
  hasPreferences="true">
```

## Attributes

**name** Required attribute. The application name.

**shortName** Optional attribute. The application's short name to be displayed in the application icon area of the Central application launcher.

**src** Required attribute. Location of the application. May be represented as an absolute URL, for example, `http://productHost.domain.com/products/myProduct/myApp/myApp.swf`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/myApp.swf`.

**background** Optional attribute. Application default background color, represented as a hexadecimal value, such as `#CCCCFF`.

**help** Optional attribute. Location of the application's help files, if any. May be represented as an absolute URL, for example, `http://productHost.domain.com/myProduct/myApp/appHelp.html`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/appHelp.html`.

**info** Optional attribute. Location of summary information about the application. May be represented as an absolute URL, for example, `http://productHost.domain.com/myProduct/myApp/appInfo.html`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/appInfo.html`.

**lang** Optional attribute. Language of the product. Currently, Central supports only the value `en`, which indicates English.

**version** Required attribute. Your application version.

**category** Optional attribute. For a description of valid categories, see [“product element” on page 397](#).

`hasPreferences` Optional attribute. Indicates whether Central should display the *Application Name* > Preferences menu item. When the `hasPreferences` attribute is set to `true`, the Preferences item appears in the *Application Name* menu. This enables the user to select the menu item, which in turn causes Central to call the `showPreferences()` function in the application. When the `hasPreferences` attribute is set to `false`, the default, the menu item does not appear in the *Application Name* menu. For more information about implementing custom preferences in an application, see [Chapter 3, “Application-specific preferences,”](#) on page 62.

# initialData element

## Description

Child element of the `application`, `agent`, and `pod` tags; user-defined content that Macromedia Central places in an object that is passed as the `initialData` argument to the `onActivate()` function in the `application`, `agent`, or `pod`. You set both the name and value of the attributes, in the following form:

```
<initialData param1="value1" param2="value2" />
```

Macromedia Central translates the `initialData` tag into an `ActionScript` object and passes it to the `onActivate` event handler.

The attributes can have any name you choose, and the values must be strings. To use a number, pass it first as a string and then convert it to a number in `ActionScript` with the `number()` method.

The following XML fragment shows how `initialData` values can be defined in the `product.xml` file:

```
<agent name="BetaAppAgent" src="agent.swf">
  <initialData foo="bar" black="white" good="evil" up="down"/>
</agent>
```

This XML code would result in an object being passed to the `onActivate()` function with the following structure:

```
{
  foo: "bar",
  black: "white",
  good: "evil",
  up: "down",
}
```

## Usage

```
<initialData param1="value1" param2="value2" />
```

## Attributes

[*param\_n*] Optional attributes. `ActionScript` parameters that Macromedia Central passes to the `application`, `agent`, or `pod`'s `onActivate` event handler.

## podClass element

### Description

Child element of the `application` tag; represents a pod SWF file that the application can use if the application calls the `shellRef.addPod()` method. If the application does not dynamically display pods, you can omit this element and use the `pod` tag alone. The `pod` tag causes a pod to be displayed when the application is installed.

### Usage

```
<podClass
  name="myPod"
  src="http://www.mycompany.com/Central/myApp/myPod.swf"
  height="90"/>
```

### Attributes

`name` Required attribute. Your pod class name.

`src` Required attribute. Location of your pod SWF file. May be represented as an absolute URL, for example, `http://productHost.domain.com/myProduct/myApp/myPods/myPod.swf`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/myPods/myPod.swf`.

`height` Optional attribute. Indicates the height of the pod. Default pod height is 100 pixels.

For more information about pods, see [Chapter 4, “Creating Pods,” on page 81](#).

# pod element

## Description

Child element of the `application` tag; specifies a pod to be displayed in the Console window when the application is installed. Can be used with or without a `podClass` element.

## Usage

```
<pod
  name="myPod"
  class=""
  src="http://www.mycompany.com/Central/myApp/myPod.swf"
  enabled="1"
  height="90"
  viewed="true">
</pod>
```

The most basic configuration of a pod does not require a class and can be used if you are not dynamically creating any pods:

```
<pod
  name="QuickSearch"
  src="http://host.domain.com/myProduct/myApp/myPods/mySimplePod.swf" />
```

## Attributes

**name** Required attribute. This is the name given to the created instance of the pod SWF file.

**class** Optional attribute. The name of the class defined in the `podClass` tag that this pod is an instance of.

**src** Optional attribute. The location of your pod SWF file, used only when declaring a pod that is not an instance of a pod class. May be represented as an absolute URL, for example, `http://productHost.domain.com/myProduct/myApp/myPods/myPod1.swf`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/myPods/myPod1.swf`.

**enabled** Optional attribute. Boolean value that causes your pod to be activated when Macromedia Central starts. Acceptable values are `true` and `false`.

**height** Optional attribute. Indicates the height of the pod. The default pod height is 100 pixels.

**viewed** Optional attribute. Boolean value that indicates whether the pod will be visible (`true`) when the application is installed, or not (`false`).

For more information about pods, see [Chapter 4, “Creating Pods,” on page 81](#).

# agent element

## Description

Child element of the `application` tag; indicates whether an agent is being implemented with the application.

## Usage

The most basic configuration of an agent does not require a class:

```
<agent
  name="agent_1"
  src="http://host.domain.com/myProduct/myApp/myAgents/mySimpleAgent.swf"
  started="true" />
```

## Attributes

`name` Required attribute. Your agent instance name.

`src` Required attribute. The location of your agent. May be represented as an absolute URL, for example, `http://productHost.domain.com/myProduct/myApp/myAgents/myAgent1.swf`, or as a pathname relative to the location of the `product.xml` file, for example, `myApp/myAgents/myAgent1.swf`.

`started` Optional attribute. Boolean value that, when `true`, causes your agent to start when Macromedia Central starts. Acceptable values are `true` and `false`. The default is `true`. The following conditions affect whether an agent starts:

- If the user disables background tasks in the Central Preferences, the agent does not start.
- If the application, pod, or agent stops the agent with the `stopAgent()` method, the agent does not start until `startAgent()` is called, even if the application or Central restart.

For more information about agents, see [Chapter 5, “Creating an Agent,”](#) on page 93.



## supportedTypes element

### Description

Child element of the `application` and `pod` tags; indicates the namespace, schema, and supported data types for the application or pod's implementation of the Blast feature.

The Blast feature allows applications to send complex data types, such as addresses and telephone numbers, to other applications. For more information about using the Blast feature, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

### Usage

```
<supportedTypes
  namespace="http://www.myCo.com/CentralData#"
  schema="http://www.myCo.com/CentralData.xsd">

  <type>address</type>

</supportedTypes>
```

### Attributes

`namespace` Required attribute. Unique namespace for your data type schema.

`schema` Required attribute. XML schema file used by the application to define the structure of each of its data types.

## type element

### Description

Child element of the `supportedType` tag; indicates the types of data that the application can receive from other applications that use the Blast feature for sharing data. To specify multiple data types that are supported by the application, use multiple `type` tags within the `supportedTypes` tag. Using the value `any` within the `type` tag indicates that your application can receive selected item data for all data types defined in the schema indicated in the `supportedTypes` tag.

The Blast feature allows applications to send complex data types, such as addresses and telephone numbers, to other applications. For more information, see [Chapter 7, “Using the Blast Feature,” on page 105](#).

### Usage

```
<supportedTypes
  namespace="http://www.myCo.com/CentralData#"
  schema="http://www.myCo.com/CentralData.xsd">

  <type>any</type>

</supportedTypes>
```

## Detailed product.xml example

The following is a detailed example of the contents of the `product.xml` file:

```
<product
  name="myProduct"
  vendor="My Company"
  created="12/21/02"
  productID="XXXXX-XXXXX-XXXXX-XXXXX"
  version="1"
  price="Free"
  requireVersion="1"
  category="Other">

  <description>"Internet News Reader"</description>

  <author
    name="Joe Culayd"
    info="http://www.mycompany.com"/>

  <icon
    src="http://www.mycompany.com/Central/myApp/1st_icon.swf"
    size="40"/>

  <application
    name="1st_of_many"
    src="http://www.mycompany.com/Central/myApp/1st_of_many.swf"
    version="1"
    help="http://www.mycompany.com/Central/myApp/app_help.html"
    info="http://www.mycompany.com/Central/myApp/app_info.html"
    background="#000000"
```

```

lang="en"
category="Other"
sendto="true"
hasPreferences="true">

<description>
  "This is my story....."
</description>

<file
src="http://www.mycompany.com/myStrings/myStrings.txt"/>

<author
name="Joe Culayd"
info="http://www.mycompany.com"/>

<icon
src="http://www.mycompany.com/myIcons/app_icon.swf"
size="40"/>

<supportedTypes
  namespace="http://www.myCo.com/CentralData#"
  schema="http://www.myCo.com/CentralData.xsd">

  <type>address</type>
</supportedTypes>

<initialData/>

<podClass
  name="myPod"
  src="http://www.mycompany.com/Central/myApp/myPod.swf"
  height="90"/>

<pod
  name="myPod"
  class=""
  src="http://www.mycompany.com/Central/myApp/myPod.swf"
  enabled="1"
  height="90">

  <file
  src="http://www.mycompany.com/myStrings/myStrings.txt"/>

  <initialData/>

  <supportedTypes
    namespace="http://www.myCo.com/CentralData#"
    schema="http://www.myCo.com/CentralData.xsd">

    <type>address</type>
  </supportedTypes>
</pod>

```

```
<agent
  name="myAgent"
  src="http://www.mycompany.com/Central/myApp/myAgent.swf"
  started="true">

  <initialData/>

</agent>
</application>
</product>
```

# INDEX

## Symbols

`_global` variables 123, 137  
`_level0` identifiers 122, 136  
`_lockroot` 36  
`_lockroot` property 122, 136  
`_root` 36, 123  
`_root` identifiers 36, 122, 136

## A

ActionScript 2.0 34  
additional resources 9  
agent files 27  
Agent Manager 20, 22  
Agent object 137  
`agent.onActivate`, best practice 138, 174  
`agentData` object 196, 348  
AgentManager object 144  
agents 23  
appID  
    with `Application.onActivate()` 173  
appID property  
    with `AgentManager.getNotices()` 153, 198, 351  
    with `AgentManager.getViewedApplications()` 159, 204, 359  
application  
    files 26  
    finding 131  
    icons 26  
    installation badge 128  
    installing 16  
    parts, sharing data between 30  
Application Finder 19, 131  
Application object 171  
Application SWF files 22  
`applicationRecs` structures 159, 204, 359  
applications, using qualified names 125

authentication 31  
authoring templates 121  
AuthoringExtensions.mxp 12

## B

badge 16, 128  
best practices 121, 162, 207, 362  
Blast data, sending 335  
Blast feature 30, 76, 209, 337, 358, 373, 375  
Blast, `SelectedItem` 335

## C

cache 29  
caching 64  
caching files 162, 207, 362, 402  
Central  
    controller 19  
    environment, elements of 20  
    framework 19  
    new features in 1.5 34  
    Player 19  
    shell 20, 21  
Central object 184  
Central Product Setup Wizard 13, 129  
`Central.DataProviderClass` object 218  
`Central.LCDataProvider` object 271  
class  
    attaching to a symbol 39  
classes, intrinsic  
    installing 12  
`clearInterval()` method 123  
close 142, 177, 320  
code, unused 125  
coding conventions 122  
collapsed property 160, 205, 360  
communicating 47

- components 125
  - installing 12
  - new names 35
  - version 2 34
  - viewing 13
  - with ActionScript 2.0 35
- compression, turning off 125
- configuration information 19
- configuring Flash 121
- connection objects 57
- Console 20, 22
  - communication with a pod 90
- Console object 186
- context menus 75
- Converting existing Flash applications 126
- coordinates, passing 123
- cross-domain access 77

## D

- data 47
  - caching locally 64
  - sharing 30
- data access 28
- data provider, specific to Central 271
- data storage 28
- data typing 19
- deactivation 41
- debug panel 13
  - installing 12
- debugging 17
- deploying files 121
- detail 312, 390
- dismissed notice 42
- displaying pods 82
- domains 77
- downloading files to Central 30

## E

- element 312, 390
- elements, Central environment 20
- engage 142, 177, 320
- event object 142, 177, 320
- events 37

## F

- faultactor property 312, 390
- faultcode property 312, 390
- faultstring property 312, 390

## file

- caching size limit 402
- download limit 402
- file element
  - caching limit 402
- file types
  - adding to local Internet cache 151, 193, 346
  - local Internet cache 162, 207, 362
- FirstApp 13
- Flash API Deltas 136
- Flash Player version 11, 34
- Flash XML schema APIs 30
- font definitions, avoiding 125
- fonts embedded in text 125
- frame rate 121

## G

- Generate Size Report command 125
- getBounds() command 44
- getMaximumSize() function deprecated 36
- getMinimumSize() function 42
- gShell variable 174

## H

- hitTest() method 123
- hosts within the same domain 151, 162, 194, 207, 346, 362
- HTTPS 31

## I

- icons 26
- id property 148, 155, 191, 200, 343, 353
- initialData 142, 174, 178, 321
- initialData property 147, 153, 189, 198, 342, 351
- initialization calls 36
- installation badge 13, 128
- installing applications 16
- intended audience 8
- isConnected() 64

## L

- layout manager 45
- LCDataProvider object 47
- LCService object 47, 295
- local caching 19
  - file size limit 402
- local connection 30
- local data 64
- local data storage 29

- local Internet cache 29
- local Internet files 29
- local shared objects 29
- LocalConnection objects 57
- Log object 300

## M

- Macromedia Central environment 20
- Macromedia Central, installing 11
- Macromedia Exchange Manager, installing 12
- Macromedia Flash Player
  - Download Center 12
  - Installation page 12
- managing object 174
- MD5 object 303
- menus 75
- minimum size 42
- mouse 123
- MovieClip object 305
- mx.central.Application interface 37, 38
- mxp file 12

## N

- network 19
- network change 41
- network connection, checking for 123
- network status 64
- notice event 42
- noticeData property 153, 198, 351
- noticeData structure 142, 146, 153, 177, 189, 198, 320, 341, 352
- NoticeID property 147, 189, 342
- Notices 30

## O

- objects
  - Log 300
  - PendingCall 306
- onActivate() function 28, 39, 173, 184, 185
- onDeactivate() function 41, 123, 124
- onFault () function 390
- onFault() function 332
- onNetworkChange() function 41, 64, 174, 317
- onNoticeEvent() function 42, 141, 176, 319
- onNoticeEvent.event() function 142, 177, 320
- onNoticeEvent.initialData 142, 178, 321
- onResize() function 41, 45
- onResult() function 31
- onUninstall() function 43, 125

## P

- passing data 47
- PendingCall object 306
- pod
  - default height 149, 156, 191, 201, 344, 354
  - width 149, 156, 191, 201, 344, 354
- Pod object 314
- Pod.onActivate, best practice 317
- podClass 149, 156, 191, 201, 344, 354
- podClass tag 84
- podData parameter 148, 155, 191, 200, 343, 353
- podData property 160, 205, 360
- PodData.height 149, 156, 191, 201, 344, 354
- PodData.name 148, 155, 191, 200, 343, 353
- podID property 149, 192, 344
- pods
  - about 23
  - communication with Console 90
  - dimensions 81
  - displaying 82
  - files 27
  - getting properties 87
  - properties 83
  - removing 86
  - UI 23
  - working with functions 88
- Portable Executable formats 151, 162, 193, 207, 346, 362
- position property 160, 205, 360
- preferences
  - data 43
  - working with 57
- product
  - definition in Central 19
  - programmatic flow 28
- product ID 16, 127
- product.xml file 20, 128
- product.xml tags, number allowed 396
- product.xml, sample 395
- products 20
- programmatic flow 28
- publishing an application 129
- publishing tool, installing 12

## R

- RegExp object 325
- regular expressions 325
  - in Central 31
  - native support 31

- remote data 30
- remove 142, 177, 320
- resize screen behavior 36
- resizing
  - the applicaiton window 44
- resizing the application window 41, 45
- root of application window 36
- routing of calls 19
- RPC object 331

## S

- sample files 13
- samples 36
- SDK, installing 12
- section2 326
- section3 137
- SelectedItem 30
- SelectedItem object 335
- services 68
- setBaseTabIndex parameter 173
- setInterval() method 123
- setSize() method 45
- setup wizard 129
- sharing data
  - about 30
  - across applications 30
- shell 20
  - callback object 173
- shell API 44
- shell instances 159, 204, 359
- Shell object 338
- shell size 44
- shellID property 159, 173, 204, 359
- showing pods 82
- showPreferences() 43
- shutting down an application 41
- SOAP protocol 30
- SOAP-based web services 68
- SOAPCall object 381
- SOAPFault 312, 390
- sort, optimizing 235, 236, 292
- SSL protocol 31
- Stage.height property 122, 136
- Stage.width property 122, 136
- status information 46
- storing data locally 29
- String object 383
- String.replace() method 383
- SWD files 125
- SWF files, optimizing 125

- symbol
  - special 39
- system requirements 11

## T

- testing 17
- timeout 142, 177, 320
- trace() commands 125
- trace() statements 18, 137
- typographical conventions 9

## U

- uninstalling an application 43
- unsafe file types 151, 162, 193, 207, 346, 362
- updates 30
- updating the user 30
- user interface design 122

## V

- viewerID property 160, 205, 360

## W

- web services 68
- WebService object 30, 384
- window 44
- workflow 25
- WSDL protocol 30

## X

- XML object 392
- XML schema, Blast 30
- XML-RPC file 334
- XML-RPC web services 73
- XML.setType() method 392