

# **Netcaster Developer's Guide**

Netscape Communicator

25 September 1997

Netscape Communications Corporation ("Netscape") and its licensors retain all ownership rights to this document (the "Document"). Use of the Document is governed by applicable copyright law. Netscape may revise this Document from time to time without notice.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. IN NO EVENT SHALL NETSCAPE BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, ARISING FROM ANY ERROR IN THIS DOCUMENT.

The Document is copyright © 1997 Netscape Communications Corporation. All rights reserved.

Portions copyright © 1996 Marimba, Inc.

Netscape, Netscape Navigator, and the Netscape N logo are registered trademarks of Netscape Communications Corporation in the United States and other countries. Other Netscape logos, product names, and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product or brand names are trademarks of their respective owners.

The downloading, export or reexport of Netscape software or any underlying information or technology must be in full compliance with all United States and other applicable laws and regulations. Any provision of Netscape software or documentation to the U.S. Government is with restricted rights as described in the license agreement accompanying Netscape software.



Recycled and Recyclable Paper

©Netscape Communications Corporation 1997

All Rights Reserved

Printed in USA

97 96 95 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

# Contents

<b>Preface</b> .....	ix
Purpose of This Document .....	ix
Structure of This Document .....	ix
Typographic Conventions .....	x
<b>Chapter 1 Channels</b> .....	13
The Channel Concept .....	13
How Channels Differ From Ordinary Web Sites .....	14
What is Webtop Mode? .....	15
What to Put in a Webtop .....	15
<b>Chapter 2 Developing Channels</b> .....	17
Channel Elements .....	17
User-Interface Design .....	18
Language Features .....	18
Design Process .....	19
Design Recommendations .....	20
<b>Chapter 3 Interacting with Netcaster</b> .....	25
Netcaster Component .....	25
Channel Object .....	26
Importing Methods .....	26

Subscription Process .....	26
Netcaster Properties .....	27
active .....	27
componentVersion .....	27
name .....	28
Netcaster Methods .....	28
activate .....	28
addChannel .....	28
getChannelObject .....	29
Channel Object Properties .....	29
absoluteTime .....	29
cardURL .....	30
depth .....	31
desc .....	31
estCacheSize .....	31
heightHint .....	32
intervalTime .....	32
leftHint .....	33
maxCacheSize .....	33
mode .....	33
name .....	34
topHint .....	35
url .....	35
widthHint .....	35
Sample Function .....	36
Caching Behavior .....	37
Redirection and Aliases .....	38
Uncached Items .....	39
Java Applets .....	39
Streamed Content .....	39
Robots Control Specification .....	40
Debugging Caching Behavior .....	42

<b>Chapter 4 Sample Channel</b> .....	47
Floating Palettes .....	47
Full-Screen Immersion .....	48
Drag-and-Drop User Customization .....	49
Animation .....	50
Automated Refresh .....	51
Context-Sensitive Help .....	51
Persistence .....	51
<b>Chapter 5 Using JavaScript with Netcaster</b> .....	53
Features Requiring Signed Code .....	54
Compatibility With Earlier Versions .....	54
Operators .....	55
Equality Operators .....	55
Delete Operator .....	55
delete .....	55
Properties .....	56
Navigator Properties .....	56
language .....	56
platform .....	57
Window Properties .....	57
innerHeight .....	57
innerWidth .....	58
locationbar .....	58
menubar .....	58
outerHeight .....	59
outerWidth .....	59
pageXOffset .....	60
pageYOffset .....	60
personalbar .....	60
screenX .....	61
screenY .....	61
scrollbars .....	62
statusbar .....	62

toolbar .....	62
Methods .....	63
back .....	63
captureEvents .....	63
charCodeAt .....	64
clearInterval .....	64
disableExternalCapture .....	64
enableExternalCapture .....	65
find .....	65
forward .....	66
fromCharCode .....	66
getSelection .....	66
handleEvent .....	66
home .....	67
moveBy .....	67
moveTo .....	67
open (window object) .....	68
print .....	70
releaseEvents .....	71
resizeBy .....	71
resizeTo .....	71
routeEvent .....	72
scrollBy .....	72
scrollTo .....	72
setInterval .....	73
setTimeout .....	74
stop .....	74
substr .....	75
toString .....	75
Objects .....	76
New Objects .....	76
event .....	76
screen .....	77

Using Literal Syntax .....	77
Events .....	79
Click .....	79
DblClick .....	79
DragDrop .....	80
KeyDown .....	80
KeyPress .....	81
KeyUp .....	81
MouseDown .....	82
MouseMove .....	82
MouseOut .....	83
MouseOver .....	83
MouseUp .....	83
Move .....	84
Resize .....	84
<b>Chapter 6 Security Considerations .....</b>	<b>87</b>
Netscape Capabilities Model .....	87
Object-Signing Protocol .....	88
Authentication Process .....	89
Certificates and Digital Signatures .....	89
Using the JAR Packager .....	90
Java Archives .....	90
Signing Scripts .....	90
Requesting Expanded Privileges .....	91
Actions Requiring Signed Code .....	92
Usage Rules for Signed Code .....	93
<b>Chapter 7 Castanet Channels .....</b>	<b>95</b>
What are Castanet Channels? .....	95
Software Applications As Well As HTML .....	96
Optimized Delivery of Updates .....	96
User Feedback, Logging, and Personalization .....	96
Scalable Distributed Server Architecture .....	97
Transactional Integrity .....	97

User's Perspective .....	97
Developer's Perspective .....	98
In General .....	98
Creating an HTML Channel .....	99
Creating an Application Channel .....	99
Creating an Applet Channel .....	99
Feedback and Personalization .....	100
Castanet Channel Extensions for Netcaster .....	101
Controlling Castanet Channels .....	101
Logging Link Clicks and Image Displays .....	102
Castanet Features and Netcaster .....	103
For More Information .....	103
<b>Glossary</b> .....	105
<b>Index</b> .....	109

This preface explains the purpose and structure of this guide and explains its typographic conventions.

## Purpose of This Document

This guide is intended for content developers who wish to create channels for Netscape Netcaster, a component of Netscape Communicator. Channels present a new type of web content combining rich media with timely information, optionally displayed in full-screen webtop mode. The guide contains technical descriptions of channel concepts, design and development processes, sample code, and the JavaScript features that implement channel capabilities.

This document assumes that you are familiar with Netcaster. If you are not, use the Help menu in Communicator, choose Help Contents, and click About Netcaster.

This guide documents Netscape Netcaster version 1.0, which is a component of Netscape Communicator version 4.02.

## Structure of This Document

This document contains the following chapters in addition to this introduction:

Chapter 1, “Channels,” explains the concepts of channels and the special type of channel called a webtop.

Chapter 2, “Developing Channels,” provides a description of the channel creation process.

Chapter 3, “Interacting with Netcaster,” describes how to use the API that enables the channel subscription process.

Chapter 4, “Sample Channel,” presents sample code that implements important channel features.

Chapter 5, “Using JavaScript with Netcaster,” describes certain JavaScript extensions built into Communicator that you can use to build compelling channels and webtops.

Chapter 6, “Security Considerations,” discusses the Netscape security model, the actions requiring signed code, and how to sign code.

Chapter 7, “Castanet Channels,” describes the Marimba Castanet technology which is integrated into Netcaster.

This guide also includes a glossary of Netcaster-related terms and a subject index.

## Typographic Conventions

Certain typographic conventions are used in this guide to indicate the category to which special terms belong. The following conventions are used throughout this guide:

- Code identifiers that express literal JavaScript and HTML syntax appear in a monospaced font like this: `computer voice`.
- Terms that are defined in the glossary appear on first introduction in a boldface typeface like this: **glossary term**.
- Italic font is used for emphasis and to indicate a special term like this: *special term*.

The following conventions are used in chapters containing JavaScript syntax descriptions:

- Code variables that represent in code placeholders that you would replace with actual values appear in a slanted monospaced font like this: *computer voice variable*.
- In syntax definitions, optional terms appear within square brackets like this: `[optional]`. The syntax of JavaScript arrays provides an exception to this convention; actual square brackets enclose the elements of an array.

- In syntax definitions, choices are separated by a vertical bar like this:  
choice1 | choice2.



# Channels

This chapter explains the concept of channels. It defines the channel concept and explains how channels differ from ordinary web sites. It also explains the concept of the special type of channel called a webtop and describes the type of content appropriate for webtop presentation.

## The Channel Concept

Channels are information-oriented web sites that use Netscape Netcaster to bring focused, up-to-date content to the user's desktop. Channels deliver dynamic information, which is transparently downloaded in the background, so that it is instantly available, online or off.

Netcaster channels can be delivered in two ways: as standards-based web-server channels or Castanet Transmitter channels. This chapter discusses web-server channels. Chapter 7, "Castanet Channels," describes Castanet channels, outlining the benefits of using a dedicated push server.

A channel is a set of web pages that use **push technology** to deliver their content automatically to the user's computer in a predetermined manner, rather than being pulled in by an explicit user interaction. Channels have focused content, which can be media-rich, persistent, and automatically updated (requiring no manual reloading). Users can customize what, when, and how often channel information gets pushed to their desktop.

Because they are built using existing web building blocks, such as HTML and JavaScript, Netcaster web-server channels provide a standards-based, open, netcasting solution that leverages existing web content and infrastructure.

## How Channels Differ From Ordinary Web Sites

Channels are like regular web sites in that they are built with HTML and JavaScript. They can include Java applets and other features. However, channels differ from web sites because instead of actively browsing to the site whenever they want some information, users subscribe to a channel, and the desired information is pushed to the user's browser at predefined intervals. In this way, the user receives up-to-date information when it becomes available, without needing to search for it.

Channels have the following characteristics:

- Channels can be updated periodically without user intervention.
- Channel content can be personalized so users get just the information they want.
- Channel information is cached, enabling richer content to be used.
- The type of content best suited for presentation via a channel is different from regular web content.

Any web site content can be delivered as a channel, enabling you to leverage your existing web content; however, dynamic information presented through rich media takes best advantage of Netcaster's channel technology. Channels that run in webtop mode have additional characteristics.

## What is Webtop Mode?

A webtop is a type of channel that is anchored to the desktop; that is, a webtop locks its content to the backmost layer of the client's display (of the Communicator layer in Mac OS and Unix). Webtops usually cover the entire screen. The main idea of the webtop concept is that you display your content in the desktop view and are the user's primary experience.

A webtop can be defined as a particular kind of web content displayed in a fixed manner behind the user's regular application windows. Users can choose a favorite channel to set as their webtop. Because webtop content is always present, you will want to provide compelling content that users will find useful, as well as visually attractive.

Channels that run in webtop mode have these characteristics:

- Webtops usually cover the user's full display screen.
- Webtops are locked to the backmost layer of the display (behind all windows in the Windows environment, and behind all Communicator windows on Macintosh and Unix).
- Links from webtops should open new windows, rather than changing the webtop.

## What to Put in a Webtop

All channels provide pushed information, but channels running in webtop mode deserve special consideration. The technical differences between webtops and other types of channels are minor: channel content is displayed in a browser window, whereas webtops are immersive, covering all or much of the display screen. While all channels bring media-rich information to the user, webtops provide a workspace for interacting with web content. The main differences between channels and webtops lie in the kinds of information that are appropriate to them.

Information that is well suited to the webtop is that which is of continuous interest to the user. The content can be dynamic in the sense that the details may change often, but the information should always be relevant and up to

date. For example, a stock ticker or news wire service present constantly changing particulars, but the user desires the presence of that information to be continuous.

Examples of information types suitable for webtops are

- tidbits and highlights that link off to the full story elsewhere
- dynamic information perceived to be “live,” such as a stock ticker
- services that enable users to request information delivery, such as package trackers and search engines
- webtop information that can be passively monitored at the backmost layer while users perform their daily computing activities

A webtop should present launch points to detailed information. The information on the webtop should not require scrolling. Instead of loading new content into the webtop, a launch point should always spawn a new window and load the new content into it. These windows open in front of the webtop and obscure its contents.

The user interface of the webtop is left entirely to the content provider. The information itself becomes the interface. Because the webtop is always present on the user’s display screen, an uncluttered, elegant design is more likely to encourage the user to remain in your webtop environment.

More information about webtop design is presented in Chapter 2, “Developing Channels.”

# Developing Channels

This chapter describes the process of developing channels. It discusses the elements and languages from which channels are built and the special considerations they require, such as principles of user-interface design. This chapter also presents a suggested channel design process and techniques to optimize channel performance and effectiveness.

## Channel Elements

Channel development is technically the same as developing a regular web site in that channels are built from HTML, JavaScript, and Java applets. However, channels require special consideration because of their particular characteristics.

A channel is built primarily from HTML and JavaScript files. There is one HTML file at the top-level URL that is the channel address. Typically, the top-level HTML file is linked to other HTML files composing the site. These HTML files describe the layout of every screen displayed to the user.

HTML files usually contain any textual content appearing on the pages they describe. They contain tags that refer to separate graphics and other media files, such as GIF format files, to be displayed in the layout.

The other major component of a channel is a set of JavaScript files, referred to by tags in the HTML files. The JavaScript files provide much of the dynamic capabilities of the channel. Functionality may also be provided by Java applets stored in their own files, as well as content interpreted by installed plug-in components.

## User-Interface Design

Designing a channel implies responsibilities for good user-interface design not incumbent on designers of regular web sites. This additional responsibility is particularly true of full-screen and webtop-mode channels, because these channels deliberately reject use of the standard menus and navigation controls of the Communicator window, as well as other desktop user-interface elements. Therefore, you must provide what the user needs to control the channel.

Webtop design demands further consideration because of the likelihood that users will use other applications while the webtop is visible. Keep in mind that the webtop should work with these applications and not compete with them for the user's attention and screen real estate. Basically, webtop content should be carefully placed so it is visible but not distracting. Keeping in mind that people will be looking at this information for hours at a time, design the webtop so the user will find its presentation appealing, even after extended periods of use.

Possibly the most important thing to realize about channel development is that you are building software. You are building from scratch an interface onto the web, unconstrained by any browser or desktop interface. Netcaster blurs the distinction between content and program. You are creating not only a presentation of information, but you are also creating the presentation software itself. In this sense, Netcaster is the development platform, providing two interpreters (for HTML and JavaScript) for layout and event handling, as well as providing services, such as network connectivity and security.

## Language Features

To provide a useful user interface and the features that make a channel compelling, use dynamic HTML, including layers and style sheets. Information about dynamic HTML is available at the following URL:

<http://developer.netscape.com/library/documentation/communicator/dynhtml/index.htm>

Many of the fundamental features of channels and webtops use the new features in JavaScript 1.2, as described in Chapter 5, “Using JavaScript with Netcaster.” In particular, JavaScript 1.2 enables you to retrieve properties of the client environment and to create an interactive user interface for your channel.

## Design Process

The design process for every channel is as unique as its requirements and design goals. Certain steps along the way are common, however. The steps in this section represent recommendations borne of experience in channel design. Sample code that implements channel features is presented in Chapter 4, “Sample Channel.”

- 1. Develop an object tree.** You may want to create a storyboard or flowchart describing the presentation sequences possible for the user. The top-level URL at the channel address represents the root of the tree. From it, you can trace the structure of the rest of the site. Generally, an HTML page correlates to a single topic or aggregation of topics within the site content.
- 2. Create a structure that optimizes downloading.** Determine which content will be persistent and which will change often. Considering these factors, you can create a download schedule for Netcaster to use to cache the channel content. This schedule is expressed in your site’s netcasting parameters.

In addition, consider that the mechanism used by Netcaster to download channels includes a parameter defining the number of levels of content to download. The top-level page and all the graphics and JavaScript or Java files it refers to are one level. All the pages referred to by the top-level page, including their graphics and other media content files and their JavaScript (and Java) files, make up the second level. You need to consider this structure when you determine the recommended site-crawling level for your site’s netcasting parameters. Netscape recommends designing your channel so that two levels of content are sufficient to download, as described in “Design Recommendations”.

3. **Develop user interface elements and container positions.** Typically, these design elements are implemented using HTML and the JavaScript event model.
4. **Evaluate your objects for modularity.** For example, ensure that no global variables will cause scoping problems. Try to design reusable components. Test everything separately, then integrate the pieces together.
5. **Determine the desired interaction behavior and visual effects.** For example, use JavaScript to animate a visual cue when the user rolls the mouse over a button or to slide layers in and out of view.  
  
Make sure that your layer structure works using passive content initially, before introducing more complex code that could introduce other bugs.
6. **Link in live content and test your design.** Remember that because HTML and JavaScript are interpreted, bugs appear at runtime; there are no compile-time syntax checks.
7. **Do user testing.** Perform a series of user tests with people having various levels of experience corresponding to those of your users.  
  
The final arbiter of any design is the user. In addition to all technical implementation steps, user testing, collection of feedback, and subsequent redesign are necessary and important elements of your development process.

Finally, implement a button or link that adds your channel to the user's My Channels container. This button or link can reside, for example, on a channel preview page. Technical details of this subscription process are described in Chapter 3, "Interacting with Netcaster."

## Design Recommendations

This section presents some channel design recommendations to optimize channel performance and effectiveness.

- **Use rich media such as sound.** Because channel content is downloaded in the background, channel providers can take advantage of rich media types. Sound is especially effective. Consider creating an opening sequence for your channel that includes animation and sound.

- **Consider using webtop mode.** Webtop presentation is optimal for channel content that is intended to be viewed passively, and it provides a focused, immersive user experience. In addition, low-resolution displays limit the amount of space available for content, and that space is further reduced by user-interface widgets such as the Navigator tool bars.
- **Include height and width attributes in all image tags.** Adding explicit size information to images enables Communicator to perform page layout much more efficiently and optimizes offline viewing. If your channel loads slowly when viewed offline, check for missing height and width attributes; cached content is local, so it should display quickly.
- **Don't use redirection and aliases.** Server-side redirection and page aliases (implicit URLs) direct Communicator to a different URL from that originally specified in a link. These actions can cause unexpected behavior when Netcaster downloads your channel. (See “Redirection and Aliases” on page 38.)
- **Design for resolution independence.** New features of JavaScript 1.2 enable you to implement content that is independent of the user's display screen resolution and formatted to take up the entire width and height of the display area.

Channel layout should be visually and functionally well behaved at standard screen resolutions across all supported platforms. Test your design in at least these three screen resolutions: 640-by-480, 800-by-600, and 1024-by-768. Optimize for 800-by-600 first, with 640-by-480 a close second.

Information about the user's display screen is accessible in JavaScript using the `screen` object (see page 77). You can format content to work correctly in either a webtop or browser window using the `window.innerWidth` and `window.innerHeight` properties (see page 57).

- **Avoid unneeded scrollbars.** Typically, channels should make the contents visible in response to other user actions or animation sequences, rather than simply leaving the user to scroll them into view. Webtops in particular

should not require scrolling to view any part of their overall layout. However, individual content containers, such as text fields, may need to scroll through their particular content.

There are two situations in which you can explicitly turn off scrollbars. In the first case, if you create layers offscreen, make them invisible initially. Otherwise, Communicator displays scrollbars to enable the user to scroll the layers into view manually.

In other cases, when you simply want to turn off scrollbars in a channel window, use the following JavaScript statements:

```
window.document.width = 0;  
window.document.height = 0;
```

Put these statements at the top of your script, and make sure the script is referenced after all the <LAYER> tags. The Netscape channel provides a good example of this technique.

The purpose of the document property of the window object is to tell Communicator explicitly the size of a window's logical image area, aggregating all of its layers and content. Usually you do this so Communicator can provide scrollbars to give the user access to hidden content. In this case, you are telling Communicator explicitly not to provide scrollbars.

Another way to avoid scrollbars on a window is to open it using the `scrollbars=no` option of the `window.open` method. However, this approach also removes programmatic control of scrolling, so it is not useful for this purpose.

- **Try not to use more than two levels of cache depth.** Each additional level of links crawled typically requires an exponential amount of resources in terms of processing power, bandwidth, and disk space. Experience has shown that two levels of caching are plenty for most usable channel topologies.
- **Use invisible anchors to download required files.** Content that is not contained in the first two levels of the channel's link hierarchy can be downloaded using invisible anchors; that is, <A HREF> tags that refer to a link URL, but contain no visible text.

Invisible anchors provide a way for you to control the behavior of the Netcaster crawler, the mechanism that downloads channel content into the Netcaster client's cache. Technically, invisible anchors are <A HREF> tags

that refer to a linked URL, but contain no visible text. For example, the following invisible anchor invisibly includes the indicated HTML file—no text content appears between the `<A HREF>` and `</A>` tags.

```
<A HREF="http://www.otherServer.com/myRandomFile.html"></A>
```

The advantage of invisible `<A HREF>` tags is that caching of files is not restricted by the link hierarchy. You can make a hypertext reference to a file anywhere on any page that is downloaded without including any text to be marked as the link. You can also include anchors in a hidden layer to enable caching of linked content. You may find it helpful to maintain a list of the files you wish to cache, and include the list in a hidden layer in your top-level channel file.

- **Maintain a consistent user experience.** Consider the behavior of your channel when viewed offline. Ensure that content needed for a consistent user experience is downloaded when Netcaster crawls the channel. For example, ensure that you cache images that create a highlight effect when the user rolls the mouse over them (using an `onMouseOver` handler).

Similarly, if you have a list of links to other files, ensure all of those files are cached and available for offline viewing. Or, if your design does not require offline availability of those files, don't cache any of them. Don't present the user with a set of links that behave inconsistently, some live and some dead.

## Design Recommendations

# Interacting with Netcaster

This chapter describes the Netcaster API (application programming interface), which is implemented in JavaScript. It also describes the flow of the subscription process, so you can see exactly where and how your channel needs to interact with Netcaster. This chapter also describes the manner in which Netcaster downloads channel content and presents ways to control and debug that behavior.

## Netcaster Component

Netscape Netcaster is represented in JavaScript as the `netcaster` component, an element of the `components` array. The `components` array provides information about the status of the `netcaster` component. The `name` and `componentVersion` properties provide the name and current version number of the `netcaster` component. The `active` property determines whether the `netcaster` component is installed and running. The `activate` method starts execution of the `netcaster` component.

## Channel Object

Channel code interacts with Netcaster through a channel object, to which you get access through the `netcaster` component. Each channel to which Netcaster is subscribed is represented by a channel object. The channel object has properties that specify the parameters for the channel. You manipulate the channel object through two methods that belong specifically to the `netcaster` component: `getChannelObject`, which creates a channel object, and `addChannel`, which completes the subscription process.

## Importing Methods

Before you can use the `netcaster` component methods, you must explicitly import them into the scope of the current script. This requirement is a feature of the JavaScript security model. First you must ensure that the `netcaster` component is active; then you must import its methods with the following JavaScript statements:

```
import components["netcaster"].getChannelObject;
import components["netcaster"].addChannel;
```

Once the methods have been imported, you can call them in the local scope without explicit reference to the `netcaster` component. See “Sample Function” for an illustration of this process in the context of a working implementation.

## Subscription Process

The subscription process occurs when a user adds a channel to Netcaster. Programmatically, it begins when a script obtains the channel object using the `getChannelObject` method, and it culminates in a call to the Netcaster component’s `addChannel` method.

The web page from which you initiate the channel subscription process should display a button or link that invokes the `getChannelObject` and `addChannel` API calls. You could place this button on your site’s home page, for example, or on a preview page for your channel, or even in an email message. See “Sample Function” for a sample implementation of the subscription process. More samples and a wizard that automatically creates JavaScript code for this purpose is available at the URL

`http://developer.netscape.com/library/examples/index.html`

When the `addChannel` call is made, Netcaster uses the values of the channel object properties as the default channel property values and presents a confirmation dialog box to the user.

The user can change any of the channel property values through the Channel Properties dialog box, invoked by clicking the Properties button in the confirmation dialog box. Any values changed by the user override those specified as channel object properties. There is no return value or other means to determine if the user changed any of the values.

A site can have additional capability to personalize a channel, but such work must be completed prior to the invocation of the `addChannel` method. The Channel Properties dialog box is the last interaction that can take place before the channel is added. There is no way to add a channel without presenting the confirmation dialog box and an opportunity to invoke the Channel Properties dialog box.

## Netcaster Properties

The `components` array has three properties that contain information about the netcaster component: `active`, `componentVersion`, and `name`.

### **active**

Property. Returns true if the component is installed and running.

**Method of** `components` array

**Syntax** `components["netcaster"].active`

### **componentVersion**

Property. Contains the version number of the component.

**Property of** `components` array

**Syntax** `components["netcaster"].componentVersion`

**Description** The `componentVersion` property returns the string `1.0` for the initial production release of the `netcaster` component.

### **name**

Property. Contains the name of the component.

**Property of** `components` array

**Syntax** `components["netcaster"].name`

**Description** The `name` property returns the string `netcaster` for the `netcaster` component.

## Netcaster Methods

The `netcaster` component has three methods: `activate`, `addChannel`, and `getChannelObject`.

### **activate**

Method. Begins execution of the component.

**Method of** `components` array

**Syntax** `components["netcaster"].activate();`

### **addChannel**

Method. Adds a channel to Netcaster.

**Syntax** `components["netcaster"].addChannel(channelObject);`

**Method of** `netcaster`

**Parameter** *channelObject* is a variable representing a channel object previously returned by the `getChannelObject` method.

**Description** The `addChannel` method completes the subscription process by adding the channel specified in the *channelObject* parameter to the My Channels container.

## getChannelObject

Method. Returns a new channel object, preinitialized with default values.

**Method of** netcaster

**Syntax** `components["netcaster"].getChannelObject();`

# Channel Object Properties

Channel objects have properties that are used to specify the channel's netcasting parameters. First, you call the `getChannelObject` method, which returns a new channel object; next you set any relevant channel object properties; last, you call the `addChannel` method to complete the subscription process. The channel properties specified in the object serve as default values; the user can override them in the Channel Properties dialog box.

Channel object properties not documented by Netscape should be considered reserved and should not be changed.

## absoluteTime

Property. Specifies the time for daily or weekly downloads of channel content.

**Syntax** `channelObject.absoluteTime = time`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`time` is an integer value representing the absolute time for downloading, in minutes after midnight on Sunday.

**Description** The `absoluteTime` property is required if the `intervalTime` property is `-5` (daily) or `-6` (weekly); otherwise it is ignored. Acceptable bounds for daily events are 0–1339 minutes; bounds for weekly events are 0–10079 minutes. Values greater than the maximum are accepted modulo the maximum, so specifying 1440 is the same as 0 for daily events.

For example, to specify Tuesday at 5:30 p.m. as a weekly event, add (2880 + 1020 + 30) for a value of 3930. The value 2880 represents Tuesday; the 1020 represents 5:00 PM, and the 30 adds another 30 minutes to the total. The following table presents numerical equivalents for the days of the week and hours of the day:

Day	Value	Hour	Value	Hour	Value
Sunday	0	Midnight	0	Noon	720
Monday	1440	1:00 a.m.	60	1:00 p.m.	780
Tuesday	2880	2:00 a.m.	120	2:00 p.m.	840
Wednesday	4320	3:00 a.m.	180	3:00 p.m.	900
Thursday	5760	4:00 a.m.	240	4:00 p.m.	960
Friday	7200	5:00 a.m.	300	5:00 p.m.	1020
Saturday	8640	6:00 a.m.	360	6:00 p.m.	1080
		7:00 a.m.	420	7:00 p.m.	1140
		8:00 a.m.	480	8:00 p.m.	1200
		9:00 a.m.	540	9:00 p.m.	1260
		10:00 a.m.	600	10:00 p.m.	1320
		11:00 a.m.	660	11:00 p.m.	1380

**See also** The `intervalTime` property.

## cardURL

Property. Specifies the URL of a graphic icon for the channel.

**Syntax** `channelObject.cardURL = "string"`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`string` is a string representing a URL address for the graphic file.

**Description** The `cardURL` property is optional. It represents the fully specified URL of an icon to be displayed in the My Channels container when the mouse is over the link to this channel. If unspecified, a default image is used. The specified document may be scaled or clipped to fill the available space.

**Note** The `cardURL` property is unused in Netcaster 1.0.

## depth

Property. Specifies the link hierarchy depth of the channel to be downloaded into the cache.

**Syntax** `channelObject.depth = value`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`value` is an integer value between 1 and 99.

**Description** The depth property is optional. To maximize performance, most channels should not specify a depth greater than 2.

## desc

Property. Specifies the descriptive text of the channel.

**Syntax** `channelObject.desc = "string"`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`string` is a string that briefly describes the channel.

**Description** The desc property is optional.

**Note** The desc property is unused in Netcaster 1.0.

## estCacheSize

Property. Specifies the estimated cache size for the channel.

**Syntax** `channelObject.estCacheSize = size`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`size` is an integer value between 1 and 1,047,527,424 representing the estimated cache size in bytes.

**Description** The `estCacheSize` property is optional. It represents the estimated amount of disk space required to store the channel's content on the user's computer. You should specify the size of the content you anticipate delivering. If specified, this value is displayed to the user, rounded up to the nearest kilobyte, on the cache panel of the Channel Properties dialog box.

Because the user is likely to use this value to determine how much disk space to allocate to your channel, you should overstate it slightly to ensure that enough space is allocated for all your content. Accordingly, this value should be less than or equal to the value given for the `maxCacheSize` property. Specify a value of -1 if you do not know the estimated size.

## heightHint

Property. Specifies the preferred height of the channel window.

**Syntax** `channelObject.heightHint = value`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`value` is an integer value representing the number of pixels.

**Description** The `heightHint` property is optional. Netcaster may ignore this value, depending on the current screen resolution, display mode, and other factors.

**Note** The `heightHint` property is unused in Netcaster 1.0.

## intervalTime

Property. Specifies the download interval time for the channel.

**Syntax** `channelObject.intervalTime = period`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`period` is a positive integer value representing the interval period, in minutes, or one of the special negative values.

**Description** The `intervalTime` property is optional. Special negative values represent predefined times: -2 is every 15 minutes; -3 is every 30 minutes; -4 is every hour; -5 is daily; -6 is weekly. Specifying 0 is invalid.

**See also** The `absoluteTime` property.

## leftHint

Property. Specifies the preferred horizontal position of the channel window, relative to the left edge of the display.

**Syntax** `channelObject.leftHint = value`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`value` is an integer value representing the number of pixels.

**Description** The `leftHint` property is optional. Netcaster may ignore this value, depending on the current screen resolution, display mode, and other factors.

**Note** The `leftHint` property is unused in Netcaster 1.0.

## maxCacheSize

Property. Specifies the maximum cache size for the channel.

**Syntax** `channelObject.maxCacheSize = size`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`size` is an integer value between 1 and 1,047,527,424 representing the estimated cache size in bytes.

**Description** The `maxCacheSize` property is optional. It represents the maximum amount of disk space that could be required to store the channel's content on the client's computer. If specified, this value is displayed to the user, rounded up to the nearest kilobyte, on the cache panel of the Channel Properties dialog box.

## mode

Property. Specifies the display mode of the channel.

**Syntax** `channelObject.mode = "string"`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

*string* is one of the three strings `window`, `full`, or `webtop`.

**Description** The `mode` property is optional. If the value of *string* is `window`, Netcaster opens the channel into a regular Communicator window.

If the value of *string* is `webtop`, Netcaster opens the channel into a chromeless window locked to the backmost layer of the display (of the Communicator layer on Mac OS or Unix). By default, the `webtop-mode` window covers the entire display screen, unless a different size and location have been specified by the `topHint`, `leftHint`, `widthHint`, and `heightHint` properties.

If the value of *string* is `full`, Netcaster opens the channel into a chromeless full-screen window, but the window is *not* locked to the backmost layer of the display. An example of a channel that might use full-screen mode is a game in which the display content provides the navigational controls and in which the user should not be able to navigate the channel any other way. `Webtops`, by contrast, always display Netcaster's navigational panel.

**Note** Users do not get to choose between `window` and full-screen mode. If you call the `addChannel` method with either of those options, the Channel Properties dialog box indicates that the channel will appear in a Navigator window. This enables you to choose between `window` and full-screen mode, depending on the requirements of your content, while preventing users from inadvertently denying themselves navigational control. By the same token, if you use full-screen mode you have a greater responsibility to provide all of the user-interface features users need to control your channel.

## name

Property. Specifies the user-visible name of the channel.

**Syntax** `channelObject.name = "string"`

**Parameters** *channelObject* is a variable representing a channel object previously returned by the `getChannelObject` method.

*string* is a string representing the name of the channel.

**Description** The `name` property is required and must be set before the channel object can be passed as a parameter with the `addChannel` method call. After the channel is added, the name appears on a tab in Netcaster's My Channels container.

## topHint

Property. Specifies the preferred vertical position of the channel window, relative to the top edge of the display.

**Syntax** `channelObject.topHint = value`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`value` is an integer value representing the number of pixels.

**Description** The `topHint` property is optional. Netcaster may ignore this value, depending on the current screen resolution, display mode, and other factors.

**Note** The `topHint` property is unused in Netcaster 1.0.

## url

Property. Specifies the URL of the channel.

**Syntax** `channelObject.url = "string"`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`string` is a string representing a complete URL address for the channel.

**Description** The `url` property is required and must be set before the channel object can be passed as a parameter with the `addChannel` method call. The URL points to the top-level HTML page of the channel.

## widthHint

Property. Specifies the preferred width of the channel window.

**Syntax** `channelObject.widthHint = value`

**Parameters** `channelObject` is a variable representing a channel object previously returned by the `getChannelObject` method.

`value` is an integer value representing the number of pixels.

**Description** The `widthHint` property is optional. Netcaster may ignore this value, depending on the current screen resolution, display mode, and other factors.

**Note** The `widthHint` property is unused in Netcaster 1.0.

## Sample Function

The JavaScript code in this section creates a channel object, sets the channel properties, and adds the channel to Netcaster. Alternatively, you can create a script similar to the one in this section by executing Netscape's Add-Channel Wizard. The code in this example and the wizard are both available at the URL:

<http://developer.netscape.com/library/examples/index.html>

The function in this example illustrates the implementation of an add-channel link, such as you should provide to enable users to subscribe to your channel.

The HTML code following the script illustrates two ways to add the channel: an image that calls the channel-adding function when the user clicks it and a form with a button that performs the same action.

You can copy and paste this sample script directly into your HTML page, such as a channel preview page. Edit the channel object properties to contain the correct default netcasting parameters for your own channel.

```
<SCRIPT LANGUAGE="JavaScript1.2">
function addMyChannel(name,url) {
  var nc = components["netcaster"];
  nc.activate();
  if(nc.active == true) {
    import nc.getChannelObject;
    import nc.addChannel;
    channel = getChannelObject();
    channel.url = (url || "URL"); //channel URL
    channel.name = (name || "Name"); //channel Name
    channel.desc = name; //channel description
    channel.intervalTime = -5;
    channel.absoluteTime = 0;
    channel.estCacheSize = -1;
    channel.maxCacheSize = 1024000;
    channel.depth = 3;
    channel.active = 1;
    channel.topHint = screen.availTop;
    channel.leftHint = screen.availLeft;
    channel.widthHint = 600;
  }
}
```

```

channel.heightHint = 391;
channel.mode="webtop";
channel.type=1; // 1 = HTTP channel; 2 = Castanet channel
channel.cardURL = ''; //reserved for future use
addChannel(channel);
}
}
</SCRIPT>

```

Here are a couple of sample buttons for calling the addMyChannel code in order to add a channel:

```

<A HREF="" onClick="addMyChannel(); return false;">
<IMG SRC="http://developer.netscape.com/one/dynhtml/images/ncnow.gif" WIDTH=117
    HEIGHT=55 BORDER=0></A>

```

```

<FORM>
<INPUT TYPE="button" VALUE="Add My Channel!" onClick="addMyChannel();">
</FORM>

```

## Caching Behavior

This section describes certain aspects of the way Netcaster stores channel content in its cache. When Netcaster is subscribed to a channel, it automatically connects to the channel URL at the intervals specified in the channel object properties describing previously in this chapter. While the connection is active, Netcaster crawls the channel. **Crawling** is the action by which Netcaster downloads the content, beginning with the top-level page and continuing through each linked page to the depth specified in the channel object properties. The content is stored in a special local cache file.

When the crawler caches a page of channel content, it first downloads all of the HTML pages to the specified depth. Then it goes back and downloads all the images contained in those pages, from top to bottom through the link hierarchy. Finally, the crawler downloads all of the embedded objects, such as Java applets and separately referenced JavaScript scripts. This behavior ensures that users have the benefit of formatted text and working links first, in case their download is incomplete.

## Redirection and Aliases

Server software can redirect a request from a client to download a page from the requested URL to a different URL.

**Note** Links to files that are redirected or aliased by the server can cause unexpected behavior, so it is important to understand the information in this section.

### Server-Side Redirection

If server software redirects a client request, Netcaster caches the page content and indexes it by the name that is returned by the server, not the name originally requested by the client. For example, if a request for a page at the URL

```
http://myCoolSite
```

is redirected to

```
http://myCoolSite/whatsNew.html
```

through a server-side redirection (300-range return code), Netcaster stores the content and indexes the page as

```
http://myCoolSite/whatsNew.html
```

Any pages that subsequently refer to the simple URL

```
http://myCoolSite
```

do not result immediately in cache hits. When it is online, Netcaster creates a connection to the server and requests the simple URL. In response, the server issues the redirect message, and Netcaster retrieves the page from the cache. When it is offline, Netcaster issues a message that it is not connected to the network.

### Page Aliases

Aliases that are set up for a page URL can have the same effect. For example, if the client specifies a URL that points to a directory but omits the page's file specifier

```
http://myCoolSite/
```

the page address defaults to

`http://myCoolSite/index.html`

which can result in the same behavior as in the case of server redirection.

## Uncached Items

If an item referenced from a page is not cached for any reason when Netcaster crawls a site, it retrieves the missed item from the network next time the channel is viewed. If Netcaster is online at that time, it attempts a new connection. When it is offline, Netcaster issues a message that it is not connected to the network.

## Java Applets

Netcaster's caching behavior with regard to Java applets is similar to that for uncached items. Unless all of the Java class files are known at the time Netcaster crawls the channel site, requests to bind to new versions of the applet's classes cause Netcaster to attempt a new network connection. Typically, all of the java class files are known at crawl time only for single-class applets and those applets relying on a single class, and installed classes such as `java.lang.*` and IFC classes.

To enable more complicated applets to be cached, aggregate all of the specified classes in a JAR file and reference that file, rather than referencing a specific class.

## Streamed Content

Streamed content, because of its very nature, cannot be cached. Streamed content is any type of file not downloaded completely by itself, such as Netscape's LiveAudio.

## Robots Control Specification

This section describes the subset of the web **robots control** capability implemented in the Netcaster crawler. Robots control is a protocol by which you can specify URLs on your web site that are not to be downloaded to the client by a crawler.

Crawlers with robots control capability read instructions from a file on the site before visiting other URLs on the site. These instructions specify which other URLs on the site, if any, contain content that should not be retrieved.

### File Location

The robots control file must be named `robots.txt` and must reside in the same directory as the top-level URL of the channel. The crawler first tries to cache the `robots.txt` file and, if the server response indicates a success, the crawler parses it and follows any instructions applicable to the crawler. If the server response indicates that `robots.txt` does not exist, the crawler assumes no instructions are available and that access to the site is not restricted by `robots.txt`.

### File Format

The format of the `robots.txt` file is a set of records, separated by blank lines. A record is a set of lines, each of which contains a colon character (:). A record starts with one or more lines beginning with the words `User-agent`. These lines specify the crawlers to which the record applies. An asterisk (\*) can be used to specify any crawler (but an asterisk has no special meaning elsewhere in the `robots.txt` file).

Following the `User-agent` line or lines is a set of zero or more lines, each of which specifies a URL of a directory or an individual file to be ignored. Lines specifying a URL to be ignored begin with the word `Disallow`, followed by a colon and the directory or file name. (The `robots.txt` specification provides optionally for lines beginning `Allow`, indicating a directory or file specifically to be retrieved. However, the Netcaster crawler ignores `Allow` lines.)

Comment lines are allowed anywhere in the file. A comment line consists of any number of space characters, followed by a hash mark (#) character, followed by the comment text, ending with the end of the line.

## Netcaster Crawler Response

The Netcaster crawler parses the `robots.txt` file and looks for a record containing the line

```
User-agent: Mozilla
```

or

```
User-agent: *
```

whichever comes first. The name comparisons are case insensitive. If no record satisfies either condition, or no records are present at all, access is unlimited. If a record is found the crawler follows the instructions in that record.

The Netcaster crawler ignores the `Allow` lines and follows instructions only in the `Disallow` lines. To evaluate if access to a URL is allowed, the crawler attempts to match the paths in the `Disallow` lines against the URLs on the server, in the order they occur in the record. The first match found is used. If no match is found, the default assumption is that the URL is allowed.

For example, a `robots.txt` file could appear as follows:

```
# /robots.txt for http://www.foo.com/
User-agent: Mozilla
Disallow: /biz/plans.html
Disallow: /secrets
```

In response to this specification Netcaster would not download the file `plans.html` in the `biz` directory, and it would not download anything in the `secrets` directory or of its any subdirectories.

A `robots.txt` file containing the record

```
User-agent: *
Disallow: /
```

would exclude all crawlers from the entire site, assuming they implement the robots control protocol.

## Matching Algorithm

The matching process compares every character specified in the record with the path portion of the URLs on the site. Matching is case-insensitive. Characters can be encoded by their hexadecimal value as a percent sign (%) followed by two hexadecimal characters. If such an encoded character is encountered, it is

decoded prior to comparison, unless it is a reserved character. The reserved characters are semicolon (;), slash (/), colon (:), at-sign (@), equal sign (=), and ampersand (&). [Table 3.1](#) illustrates the matching process.

**Table 3.1** Disallow specification and URL Matching

Record Path	URL path	Matches
/tmp	/tmp.html	yes
/tmp	/tmp/a.html	yes
/tmp/	/tmp	no
/tmp/	/tmp/a.html	yes
/a%3Cd.html	/a%3cd.html	yes
/a%2fb.html	/a%2fb.html	yes
/a%2fb.html	/a/b.html	no
/a/b.html	/a%2fb.html	no
/%7ejoe/index.html	/~joe/index.html	yes

## Debugging Caching Behavior

This section presents techniques you can use to debug the caching of your channel content.

### Discovering Files Not Cached

The following steps enable you to determine if Netcaster is not caching any of your channel content.

- 1. Delete all existing channels.** An alternate procedure is to ensure that auto-updating is turned off for all added channels. In Netcaster, click the Options button, choose the channel, click the Properties button, click the General tab, then uncheck the Update checkbox.
- 2. DeletethecontentsofyourNetcasterarchivefolder.** (See “ContentCaches”.) Keep your archive folder open and monitor it while Netcaster crawls the channel.
- 3. Add the channel.**

4. **Let Netcaster crawl the channel.** “Starting and Stopping Crawling” explains how to make Netcaster crawl your channel and determine when crawling is complete.
5. **Put Communicator into offline mode.** In Communicator, from the File menu, choose Go Offline.
6. **Clear your Communicator cache.** In Communicator, from the Edit menu, choose Preferences, then open Advanced, choose Cache, and click the buttons labeled Clear Memory Cache and Clear Disk Cache (Windows) or Clear Disk Cache Now (Mac OS).
7. **Open your Communicator cache folder and monitor it.** (See “Content Caches”.) The cache folder should be nearly empty at this point.
8. **Try to load your channel.** When you view your channel this way in offline mode, you can simply observe if any content is missing. It is easiest to debug one page at a time.
9. **Close the channel.** For browser-window channels, close the browser window; for channels in webtop mode, click the Close Webtop button.
10. **Put Communicator into Online mode.** In Communicator, from the File menu, choose Go Online.
11. **Load your channel again.** Because Communicator is now online, any missing channel content should now appear. All the files missing from Netcaster’s archive folder should be in the Communicator cache (with encoded filenames). Copy them to a separate folder before going to another channel page.
12. **Make any changes needed.** For example, you may need to make filenames explicit to ensure that these files are crawled and cached properly. (See “Redirection and Aliases”.)
13. **Verify your fixes.** Repeat these steps to ensure that all of your channel content is viewable offline and no files end up in the Communicator cache instead of the Netcaster archive folder.

## Content Caches

Communicator has two content caches: one for Netcaster, containing channel content for later viewing (online or offline), and another for Navigator, containing previously downloaded regular web pages.

The Netcaster channel content cache is maintained in a folder named `archive`. The regular Navigator cache, although user selectable, is named `cache` by default (or `Cache f` on Mac OS). On Windows the `archive` and `cache` folders are located by default at the following pathname:

```
C:\Program Files\Netscape\Users\userName\
```

On Mac OS the `archive` and `cache` folders are located by default at the following pathname:

```
System Folder/Preferences/Netscape f/
```

When Netcaster completes crawling a channel's content, it writes a file in the `archive` which lists the URLs of the channel content files that were successfully downloaded and cached. This list file is named `N.dat`, where `N` represents a number. The list file is a regular text file, which you can open with any text editor, but don't open one while Netcaster is running.

Each line in the list file contains the URL of a file and begins with a letter designating the type of file (`L` for link, `I` for image, and `R` for anything else). For example, the following lines represent downloaded channel content:

```
L>http://netcaster.netscape.com/channel/netscapeNews.html
L>http://netcaster.netscape.com/channel/info/article5.html
L>http://netcaster.netscape.com/channel/info/article4.html
```

If you find all of your channel files in your `archive`, and every file is listed in the associated `.dat` file, then your channel is being crawled successfully.

## Starting and Stopping Crawling

Netcaster automatically crawls each newly added channel after it is added. To force Netcaster to crawl a channel, click the Options button, then choose a channel, and click the Update Now button in the Netcaster container.

To determine when the crawler has finished crawling a channel, watch the red-or-green indicator “light” on the channel’s tab in the My Channels container. A green light indicates that the crawler successfully downloaded the channel; a red light indicates that an error occurred. You can also monitor the archive folder: Netcaster writes the channel’s .dat file when crawling is complete.



## Sample Channel

This chapter presents some sample JavaScript and HTML code used in the implementation of a channel that runs in webtop mode. This chapter presents only the portions of the code that implement various features of the channel.

**Note** Some of the JavaScript features used in this chapter are protected and require signed code. These JavaScript features are distinguished in Chapter 5, “Using JavaScript with Netcaster.” More information about signing code is presented in Chapter 6, “Security Considerations.”

### Floating Palettes

Floating palettes remain at the top level of the display, in front of all other windows. In this way, the functionality of the webtop is always available to the user and is never obscured by other applications.

The following code puts the Netscape home page as the topmost window in the z-order hierarchy:

```
<HTML>
<HEAD>
<TITLE>
Z Hierarchy Example
</TITLE>
<SCRIPT LANGUAGE = "JavaScript1.2">
var target=self;
```

```

var nw=null;
var pixel,color;
var width = screen.width;
var height = screen.height;
function openAlwaysRaisedWindow() {
netscape.security.PrivilegeManager.enablePrivilege("CanvasAccess");
nw = window.open("http://destiny","test13","left=210,top=10,
outerWidth=300,outerHeight=300,location=yes,titlebar=yes,
toolbar=yes,alwaysRaised=yes");
target=nw;
netscape.security.PrivilegeManager.revertPrivilege("CanvasAccess");
}
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT TYPE=BUTTON onClick="openAlwaysRaisedWindow()"
VALUE="Open New Window (alwaysRaised)">
</FORM>
</BODY>
</HTML>

```

## Full-Screen Immersion

Part of the user's perception that the webtop provides the basic workspace environment derives from the webtop window covering the entire screen. This feature is enabled by Communicator's canvas mode. Although webtops need not open in full-screen mode, doing so enables you to create an immersive environment focused around personalized information.

**Note** Most developers need not be concerned with implementing this feature directly, because Netcaster opens your channel in the mode specified in the channel object properties (assuming those values are accepted by the user). See Chapter 3, "Interacting with Netcaster."

To use canvas mode, you open a new window with the following JavaScript code. This code opens a full-screen window with no chrome (controls and border elements) and creates a page that does not display browser menus, scroll bars, or toolbars.

```

netscape.security.PrivilegeManager.enablePrivilege("CanvasAccess");
window.open("http://home.netscape.com","home",'titlebar=no,
screenX=-1,screenY=-1,toolbar=no,scrollbars=no,menubar=no
outerWidth=screen.availWidth, outerHeight=screen.availHeight,

```

```
alwaysLowered=yes');
netscape.security.PrivilegeManager.revertPrivilege("CanvasAccess");
```

## Drag-and-Drop User Customization

JavaScript layer and style sheet capabilities, combined with the JavaScript event model, enable you to create a webpage with an exact layout of content and containers. Images can be placed anywhere on the screen and their positions recorded.

The following HTML code defines a layer with an image as its contents. The layer includes a separate JavaScript file that implements its drag-and-drop capabilities.

```
<LAYER NAME="headline" LEFT=420 TOP=75>
<IMG SRC="http://developer.netscape.com/news/viewsource/images/
bnr_sub_viewsource.jpg">
<SCRIPT LANGUAGE="JavaScript1.2" SRC="dnd.js">
</SCRIPT> </LAYER>
```

The JavaScript code that implements drag-and-drop capabilities, included here as the file `dnd.js`, appears as follows:

```
var oldX, oldY;
function beginDrag(e) {
    document.captureEvents(Event.MOUSEMOVE);
    document.onMouseMove=drag;
    oldX=e.pageX;
    oldY=e.pageY;
    return false;
}
function endDrag(e) {
    document.onMouseMove=0;
    document.releaseEvents(Event.MOUSEMOVE);
    return false;
}
function drag(e) {
    moveBy(e.pageX - oldX, e.pageY - oldY);
    oldX = e.pageX;
    oldY = e.pageY;
}
document.captureEvents(Event.MOUSEUP|Event.MOUSEDOWN);
document.onMouseDown=beginDrag;
document.onMouseUp=endDrag;
```

See Chapter 5, “Using JavaScript with Netcaster,” for more information about the new JavaScript events.

## Animation

Dynamic HTML and JavaScript animation capabilities enable you to make your webtop a compelling experience for the user. You can use JavaScript to animate HTML layers, providing dramatic animated effects.

The following code illustrates a simple example of JavaScript animation:

```
<HTML>
<HEAD>
<TITLE>Example of Sliding Layers</TITLE>
<SCRIPT LANGUAGE="JavaScript1.2">
function slideLayer(lyr,xinc,yinc,inctime,xstop) {
lyr.top += yinc
lyr.left += xinc
if (((xinc > 0) && (lyr.left < xstop)) || ((xinc < 0) && (lyr.left >
xstop))) {
    setTimeout('slideLayer(document.layers["'+lyr.name+'"],'+xinc+',
+yinc+', '+incntime+', '+xstop+'),' ,incntime);
    }
}
</SCRIPT>
</HEAD>
<BODY BGCOLOR="#ffffff"
onLoad="slideLayer(document.layers['vsbar'],-4,-4,20,4)">
<LAYER NAME="vsbar" top=100 left=100 visibility=inherit>

</LAYER>
</BODY>
</HTML>
```

Documentation about dynamic HTML and JavaScript animation is available at the following URL:

<http://developer.netscape.com/library/documentation/communicator/dynhtml/index.htm>

## Automated Refresh

Part of the appeal of the web-savvy webtop is its ability to bring dynamic content to the user's personalized workspace. This is accomplished in part by settings called **netcasting parameters**.

Netcasting parameters are configuration values that control actions required by webtops and channels for scheduled polling, background downloading, and site-crawling mechanisms. You supply default netcasting parameter values through the properties of the channel object (see "Channel Object Properties").

You can also use client-pull and server-push technologies to refresh individual web pages.

## Context-Sensitive Help

Communicator's new NetHelp architecture allows you to provide media-rich, context-sensitive help information to the user. More information describing how to implement this feature is available at the following URL:

<http://home.netscape.com/eng/help/>

## Persistence

To provide a realistic webtop experience, users should be able to modify the state of the webtop display and have it appear the same the next time they log in. The webtop must save and reuse configuration information describing the arrangement of elements on the webtop. Developers must write their own JavaScript code or use cookies to implement this feature.

Persistence

# Using JavaScript with Netcaster

This chapter provides a look at JavaScript language extensions built into Communicator which enable many of the features of channels, including the ability to run in webtop mode. These features are new in JavaScript version 1.2.

More complete documentation of JavaScript 1.2 features is available online at the URL:

[http://developer.netscape.com/library/documentation/communicator/jsguide/js1\\_2.htm](http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm)

In particular, see the online documentation for information about regular expressions and control statements not covered in this chapter.

For additional information about JavaScript, see the *JavaScript Guide* at the URL:

<http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html>

The JavaScript extensions described in this document are features of particular importance to channel developers and content providers. See also Chapter 3, “Interacting with Netcaster,” which describes additional JavaScript extensions that implement the Netcaster API.

**Note** This document does not describe all of the extensions to JavaScript released in version 1.2. In addition, these features may appear on different platforms in different releases.

## Features Requiring Signed Code

For security reasons, some of these features are available only to signed JavaScript. In this chapter, the language features requiring signed code are labeled **Protected**.

See Chapter 6, “Security Considerations,” for a list of operations requiring signed code and information about how to sign JavaScript. Refer to the JavaScript documentation at the URL listed in the beginning of this chapter for more complete information about using signed scripts.

## Compatibility With Earlier Versions

As JavaScript evolves, its capabilities expand greatly. JavaScript written for Navigator 4.0 may work in Navigator 4.0 only. To ensure that users of earlier versions of Navigator avoid problems when viewing pages that use JavaScript 1.2, use the `LANGUAGE` attribute in the `<SCRIPT>` tag to indicate which version of JavaScript you’re using.

Statements within a `<SCRIPT>` tag are ignored if the browser does not have the level of JavaScript support specified in the `LANGUAGE` attribute. For example,

- Navigator 2.0 executes JavaScript code within the `<SCRIPT LANGUAGE="JavaScript">` tag. It ignores code within the `<SCRIPT LANGUAGE="JavaScript1.1">` and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.
- Navigator 3.0 executes JavaScript code within the `<SCRIPT LANGUAGE="JavaScript">` and `<SCRIPT LANGUAGE="JavaScript1.1">` tags. It ignores code within the `<SCRIPT LANGUAGE="JavaScript1.2">` tag.
- Navigator 4.0 executes JavaScript code within the `<SCRIPT LANGUAGE="JavaScript">`, `<SCRIPT LANGUAGE="JavaScript1.1">`, and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.

By using the `LANGUAGE` attribute, you can write general JavaScript code that Navigator version 2.0 and higher recognize, and you can include additional or refined behavior for newer versions of Navigator.

# Operators

This section describes changed behavior and an additional operator in JavaScript 1.2.

## Equality Operators

If you use the `<SCRIPT LANGUAGE="JavaScript1.2">` tag, Navigator interprets the equality operators (`==` and `!=`) differently from earlier versions. Briefly, the equality operators now behave as follows:

- They never attempt to convert operands from one type to another. You must convert them explicitly.
- They always compare identity of like-type operands. Operands of different types are not equal.

These changes have been introduced to simplify the language, maintain transitivity, and help avoid errors. See the JavaScript documentation referred to at the beginning of this chapter for more information about these differences.

## Delete Operator

The core JavaScript interpreter now includes a delete operator, described in this section.

### **delete**

Core operator. Deletes an object, object's property, or an element at a specified index in an array.

**Syntax** `delete objectName`  
`delete objectName.propertyName`  
`delete arrayName[index]`

**Parameters** `objectName` is the name of any existing object.

*propertyName* is the name of any property belonging to the associated object.

*arrayName* is the name of an array.

*index* is an integer representing the index of an element in an array.

**Description** If the delete operation succeeds, the `delete` operator sets the object, property, or element to undefined and returns true; otherwise, it returns false.

## Properties

This section describes new and revised properties for the `navigator` and `window` objects in JavaScript 1.2.

### Navigator Properties

This section describes new properties of the `navigator` object.

#### language

Property. Indicates the localization of Navigator currently executing.

**Syntax** `navigator.language`

**Property of** `navigator`

**Description** The `language` property is used with the JAR Manager (see the JAR Installation Manager Developer's Guide).

The value returned is in most cases a two-letter code indicating the language for which the currently executing Navigator is localized, such as `en`. In some cases, a five-character code is returned to indicate a language subtype, such as `zh_CN`.

The `language` property is read-only.

## platform

Property. Indicates the machine type for which the currently executing Navigator was compiled.

**Syntax** `navigator.platform`

**Property of** `navigator`

**Description** The `platform` property is used with the JAR Manager (see the JAR Installation Manager Developer's Guide).

The machine type the Navigator was compiled for may differ from the actual machine type due to version differences, emulators, or other reasons. Platform values are Win32, Win16, Mac68k, MacPPC, and various versions of Unix.

Web page authors would use the `platform` property to ensure that their triggers download the appropriate JAR files. The triggering page should check the Navigator version before checking the `platform` property.

JAR-installation writers would use the `platform` property to double-check that their package is being installed on an appropriate machine or, for small JARs, to decide which of several machine-specific files to install.

The `platform` property is read-only.

## Window Properties

This section describes new and revised properties of the window object.

### innerHeight

Property. Specifies the vertical dimension, in pixels, of the window's content area.

**Syntax** [`windowReference.`]innerHeight

**Parameters** `windowReference` is optional and represents either the name of a window object or one of the synonyms `top` or `parent`. If `windowReference` is omitted, `innerHeight` refers to the current window.

**Property of** `window`

**Protected** To create a window smaller than 100 by 100 pixels, set this property in a signed script.

## innerWidth

Property. Specifies the horizontal dimension, in pixels, of the window's content area.

**Syntax** [*windowReference*.]innerWidth

**Parameters** *windowReference* is optional and represents either the name of a window object or one of the synonyms `top` or `parent`. If *windowReference* is omitted, `innerHeight` refers to the current window.

**Property of** window

**Protected** To create a window smaller than 100 by 100 pixels, set this property in a signed script.

## locationbar

Property. Specifies whether the Navigator window location bar is visible or hidden.

**Syntax** [*windowReference*.]locationbar.visible = true|false

**Parameters** *windowReference* is either the name of a window object or one of the synonyms `top` or `parent`.

**Property of** window

**Description** The `visible` property of the `locationbar` object allows you to hide or show the location bar of the specified window.

**Protected** Setting the `locationbar` property requires signed code.

## menubar

Property. Specifies whether the Navigator window menu bar is visible or hidden.

**Syntax** [*windowReference*.]menubar.visible = true|false

- Parameters** *windowReference* is either the name of a window object or one of the synonyms `top` or `parent`.
- Property of** `window`
- Description** The `visible` property of the `menubar` object allows you to hide or show the menu bar of the specified window.
- Protected** Setting the `menubar` property requires signed code.

## outerHeight

Property. Specifies the vertical dimension, in pixels, of the window's outside boundary.

**Syntax** [*windowReference*.]outerHeight

- Parameters** *windowReference* is optional and represents either the name of a window object or one of the synonyms `top` or `parent`. If *windowReference* is omitted, `innerHeight` refers to the current window.
- Property of** `window`
- Protected** To create a window smaller than 100 by 100 pixels, set this property in a signed script.

## outerWidth

Property. Specifies the horizontal dimension, in pixels, of the window's outside boundary.

**Syntax** [*windowReference*.]outerWidth

- Parameters** *windowReference* is optional and represents either the name of a window object or one of the synonyms `top` or `parent`. If *windowReference* is omitted, `innerWidth` refers to the current window.
- Property of** `window`
- Protected** To create a window smaller than 100 by 100 pixels, set this property in a signed script.

## pageXOffset

Property. Contains the horizontal scrolled position of the viewed page.

**Syntax** [*windowReference*.]pageXOffset

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

**Property of** window

**Description** The `pageXOffset` property allows you to determine the current horizontal scrolled position of the specified window which is useful, for example, before using the `scrollBy` method.

This is a read-only property.

## pageYOffset

Property. Contains the vertical scrolled position of the viewed page.

**Syntax** [*windowReference*.]pageYOffset

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

**Property of** window

**Description** The `pageYOffset` property allows you to determine the current vertical scrolled position of the specified window which is useful, for example, before using the `scrollBy` method.

This is a read-only property.

## personalbar

Property. Specifies whether the Navigator window personal tool bar is visible or hidden.

**Syntax** [*windowReference*.]personalbar.visible = true|false

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

- Property of** window
- Description** The `visible` property of the `personalbar` object allows you to hide or show the personal tool bar of the specified window.
- Protected** Accessing the `personalbar` property requires signed code.
- Note that `directories` is a synonym for `personalbar`.

## screenX

Property. Specifies the horizontal position of the window from the left edge of the display screen.

- Syntax** [*windowReference*.]screenX
- Parameters** *windowReference* is either the name of a window object or one of the synonyms `top` or `parent`.
- Property of** window
- Description** The `screenX` property allows you to query and set the horizontal position of the specified window.
- Protected** Accessing the `screenX` property requires signed code.

## screenY

Property. Specifies the vertical position of the window from the top edge of the display screen.

- Syntax** [*windowReference*.]screenY
- Parameters** *windowReference* is either the name of a window object or one of the synonyms `top` or `parent`.
- Property of** window
- Description** The `screenY` property allows you to query and set the vertical position of the specified window.
- Protected** Accessing the `screenY` property requires signed code.

## scrollbars

Property. Specifies whether the Navigator window scroll bars are visible or hidden.

**Syntax** [*windowReference*.]scrollbars.visible = true|false

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

**Property of** window

**Description** The *visible* property of the *scrollbars* object allows you to hide or show the scroll bars of the specified window.

**Protected** Setting the *scrollbars* property requires signed code.

## statusbar

Property. Specifies whether the Navigator window status bar is visible or hidden.

**Syntax** [*windowReference*.]statusbar.visible = true|false

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

**Property of** window

**Description** The *visible* property of the *statusbar* object allows you to hide or show the status bar of the specified window.

**Protected** Setting the *statusbar* property requires signed code.

## toolbar

Property. Specifies whether the Navigator window tool bar is visible or hidden.

**Syntax** [*windowReference*.]toolbar.visible = true|false

**Parameters** *windowReference* is either the name of a window object or one of the synonyms top or parent.

**Property of** window

- Description** The `visible` property of the `toolbar` object allows you to hide or show the tool bar of the specified window.
- Protected** Setting the `toolbar` property requires signed code.

## Methods

This section describes the new and extended methods in JavaScript 1.2. This section does not segregate methods by the object to which they belong but lists them all alphabetically.

### back

Method. Points Navigator to the previous URL in the current history list; equivalent to the user pressing the Navigator Back button.

**Syntax** [*windowReference*].back()

**Method of** window

**Parameters** *windowReference* is the name of a window object.

### captureEvents

Method. Enables the window or document to capture events of the specified type.

**Syntax** [*objectReference*].captureEvents(*eventType*)

[*objectReference*].captureEvents(*Event.eventType1*|*Event.eventType2*)

**Method of** window, document

**Parameters** *objectReference* is the name of a window or document object.

*eventType* is the type of event to be captured. The available event types are listed with the event object description on page 76. Note that the vertical bar is an actual part of the multiple-event syntax which is used to separate the event types to be captured.

**Description** If you need to capture events from pages or frames loaded from different servers, you must call the `enableExternalCapture` method first.

**See also** The `enableExternalCapture` and `disableExternalCapture` methods.

## charCodeAt

Method. Returns a number indicating the ISO-Latin-1 codeset value of the character at the given index.

**Syntax** `string.charCodeAt([index])`

**Method of** `string`

**Parameters** `string` is a text string or property of an existing object.

`index` is an optional parameter. It can be an integer between 0 and 1 less than the string length, or a property of an existing object. The default value is 0.

**Description** The ISO-Latin-1 codeset ranges from 0 to 255. Code positions 0 through 127 match the ASCII character set exactly.

## clearInterval

**Syntax** `clearInterval(intervalID)`

**Method of** `frame`, `window`

**Parameters** `intervalID` is an identifier of a timeout setting returned by a previous call to the `setInterval` method.

**See also** The `setInterval` and `setTimeout` methods.

## disableExternalCapture

Method. Disables external event capturing.

**Syntax** `disableExternalCapture()`

**Method of** `window`

**Description** The `disableExternalCapture` method complements the `enableExternalCapture` method, which allows scripts to capture events from pages or frames loaded from different servers.

**See also** The `enableExternalCapture` and `captureEvents` methods.

## **enableExternalCapture**

Method. Enables external event capturing.

**Syntax** `enableExternalCapture()`

**Method of** `window`

**Description** The `enableExternalCapture` method allows scripts to capture events from pages or frames loaded from different servers. Use this method before calling the `captureEvents` method.

**Protected** The `enableExternalCapture` method must be used in a script that requests `UniversalBrowserWrite` privileges.

**See also** The `disableExternalCapture` and `captureEvents` methods.

## **find**

Method. Finds the specified text string in the contents of the specified window.

**Syntax** [`windowReference`].`find`(["*string*"][,`true|false`][,`true|false`])

**Method of** `window`

**Parameters** *windowReference* is the name of a window object.

*string* is the text string searched for.

**Returns** `true` if the string is found; otherwise `false`.

**Description** If the *string* parameter is not specified, the method displays the Find dialog box, allowing the user to enter the search string. When the string is specified, the browser performs a case-insensitive, forward search.

The optional Boolean parameters must both be specified to use either. The first Boolean parameter, if `true`, makes the search case sensitive. The second Boolean parameter, if `true`, makes the search backwards.

## forward

Method. Points Navigator to the next URL in the current history list; equivalent to the user pressing the Navigator Forward button.

**Syntax** `[windowReference.]forward()`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

## fromCharCode

Method. Returns a string of characters represented by the specified sequence of codeset values.

**Syntax** `String.fromCharCode(sequence)`

**Method of** string

**Parameters** *sequence* is a series of one or more integers representing codeset values in the ISO-Latin-1 character set.

**Description** The `fromCharCode` method is a static method that returns a string value.

## getSelection

Method. Returns a string containing the text of the current selection.

**Syntax** `documentReference.getSelection()`

**Method of** document

**Parameters** *documentReference* is the name of a document object.

## handleEvent

Method. Invokes the handler for the specified event.

**Syntax** `objectReference.handleEvent(event)`

**Method of** objects with event handlers

**Parameters** *objectReference* is the name of an object.

*event* is the name of an event for which the specified object has an event handler.

## home

Method. Points Navigator to the URL specified in preferences as the user's home page; equivalent to the user pressing the Navigator Home button.

**Syntax** `[windowReference.]home()`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

## moveBy

Method. Moves the specified window by the specified amounts.

**Syntax** `[windowReference.]moveBy(horizontal,vertical)`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

*horizontal* is an integer representing the number of pixels by which to move the window horizontally.

*vertical* is an integer representing the number of pixels by which to move the window vertically.

**Protected** Moving a window to an offscreen position requires signed code.

## moveTo

Method. Moves the top-left corner of the specified window to the specified screen coordinates.

**Syntax** `[windowReference.]moveTo(xCoordinate,yCoordinate)`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

*xCoordinate* is an integer representing the left edge of the window in screen coordinates.

*yCoordinate* is an integer representing the top edge of the window in screen coordinates.

**Protected** Moving a window to an offscreen position requires signed code.

## open (window object)

Method. Opens a new web content window.

**Syntax** `[windowVar = ][window.]open("URL", "windowName", ["windowFeatures"])`

**Method of** window

**Parameters** *windowVar* is the name of a new window. Use this variable when referring to a window's properties, methods, and containership.

*URL* specifies the URL to open in the new window.

*windowName* is the window name to use in the TARGET attribute of a FORM or <A> tag. *windowName* can contain only alphanumeric or underscore characters.

*windowFeatures* is a comma-separated list of any of the following options and values:

```
toolbar[=yes|no] | [=1|0]
location[=yes|no] | [=1|0]
directories[=yes|no] | [=1|0]
status[=yes|no] | [=1|0]
menubar[=yes|no] | [=1|0]
scrollbars[=yes|no] | [=1|0]
resizable[=yes|no] | [=1|0]
hotkeys[=yes|no] | [=1|0]
innerWidth=pixels
innerHeight=pixels
outerWidth=pixels
outerHeight=pixels
screenX=pixels
screenY=pixels
alwaysRaised[=yes|no] | [=1|0]
alwaysLowered[=yes|no] | [=1|0]
z-lock[=yes|no] | [=1|0]
titlebar[=yes|no] | [=1|0]
dependent[=yes|no] | [=1|0]
```

Options that take Boolean values are true if a value of *yes* or *1* is specified. Option values are all false by default, except *hotkeys* and *titlebar*, which are true by default. You may use any subset of these options. Separate options with a comma. Do not put spaces or end-of-lines between the options. The *windowFeatures* are:

*toolbar* if true, creates a window with the standard Navigator toolbar, with buttons such as Back and Forward.

*location* if true, creates a window with the Location entry field.

*directories* if true, creates a window with the standard Navigator directory buttons.

*status* if true, creates a window with the status bar at the bottom.

*menubar* if true, creates a window with the menu bar at the top.

*scrollbars* if true, creates horizontal and vertical scrollbars when the document grows larger than the window dimensions.

*resizable* if true, creates a window with the control that allows the user to resize the window.

*hotkeys* if false, disables most hot keys (keyboard shortcuts) in a new window that has no menu bar. The security and quit hot keys remain enabled. **Protected.** Setting the *hotkeys* option to false requires signed code.

*innerWidth* specifies the horizontal dimension, in pixels, of the content area of the window. **Protected.** Resizing a window to a size smaller than 100-by-100 pixels requires signed code.

*innerHeight* specifies the vertical dimension, in pixels, of the content area of the window. **Protected.** Resizing a window to a size smaller than 100-by-100 pixels requires signed code.

*outerWidth* specifies the horizontal dimension, in pixels, of the outside boundary of the window. **Protected.** Resizing a window to a size smaller than 100-by-100 pixels requires signed code.

*outerHeight* specifies the vertical dimension, in pixels, of the outside boundary of the window. **Protected.** Resizing a window to a size smaller than 100-by-100 pixels requires signed code.

`screenX` is the distance, in pixels, the window is placed from the left side of the display screen. **Protected.** Placing a window offscreen requires signed code.

`screenY` is the distance, in pixels, the window is placed from the top of the display screen. **Protected.** Placing a window offscreen requires signed code.

`alwaysRaised` if true, creates a window that always floats on top of other windows, whether it is currently active or not. **Protected.** Setting the `alwaysRaised` option to true requires signed code.

`alwaysLowered` if true, creates a window that is always beneath other windows and does not rise above them when activated. **Protected.** Setting the `alwaysLowered` option to true requires signed code.

`z-lock` if true, creates a window that does not rise above other windows when activated. **Protected.** Setting the `z-lock` option to true requires signed code.

`titlebar` if false, creates a window without a titlebar. **Protected.** Setting the `titlebar` option to false requires signed code.

`dependent` if true, creates a window that is the child of the current window (from which the new window was opened). **Protected.** Setting the `dependent` option to true requires signed code.

See the JavaScript documentation referred to at the beginning of this chapter for more information about the `open` method.

## print

Method. Prints the contents of the active window (or frame); equivalent to the user pressing the Navigator Print button.

**Syntax** `[windowReference.]print()`

**Method of** `window`

**Parameters** `windowReference` is the name of a window object.

## releaseEvents

Method. Sets the window, document, or layer to ignore events of the specified type, sending them to the objects further along the event hierarchy.

**Syntax** `[objectReference].releaseEvents(eventType)`

`[objectReference].releaseEvents(Event.eventType1|Event.eventType2)`

**Method of** window, document

**Parameters** *objectReference* is the name of a window, document, or layer object.

*eventType* is the type of event to be captured. Note that the vertical bar is an actual part of the multiple-event syntax which is used to separate the event types to be ignored.

## resizeBy

Method. Resizes the window by the specified amounts by moving its bottom-right corner.

**Syntax** `[windowReference].resizeBy(horizontal,vertical)`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

*horizontal* is an integer representing the number of pixels by which to move the bottom-right corner of the window horizontally.

*vertical* is an integer representing the number of pixels by which to move the bottom-right corner of the window vertically.

**Protected** Resizing a window smaller than 100-by-100 pixels requires signed code.

## resizeTo

Method. Resizes the window to the specified outer width and height by moving its bottom-right corner.

**Syntax** `[windowReference].resizeTo(outerWidth,outerHeight)`

**Method of** window

- Parameters** *windowReference* is the name of a window object.
- outerWidth* is an integer representing the window's outer width in pixels.
- outerHeight* is an integer representing the window's outer height in pixels.
- Protected** Resizing a window smaller than 100-by-100 pixels requires signed code.

## routeEvent

Method. Passes a captured event along the normal event hierarchy.

**Syntax** [*objectReference*.]routeEvent(*event*)

**Method of** window, document

- Parameters** *objectReference* is the name of a window, document, or layer object.
- event* is the name of the event to be routed.

**Description** If a sub-object (document or layer) is also capturing the event, it is sent to that object. Otherwise, it is sent to its original target.

## scrollBy

Method. Scrolls the viewing area of the window by the specified amount.

**Syntax** [*windowReference*.]scrollBy(*horizontal*,*vertical*)

**Method of** window

- Parameters** *windowReference* is the name of a window object.
- horizontal* is an integer representing the number of pixels by which to scroll the viewing area horizontally.
- vertical* is an integer representing the number of pixels by which to scroll the viewing area vertically.

## scrollTo

Method. Scrolls the viewing area of the window so that the specified page coordinates become the top-left corner.

**Syntax** `[windowReference.]scrollTo(xCoordinate,yCoordinate)`

**Method of** window

**Parameters** *windowReference* is the name of a window object.

*xCoordinate* is an integer representing, in pixels, the x-coordinate on the page of the point scrolled to the top-left corner of the viewing area.

*yCoordinate* is an integer representing, in pixels, the y-coordinate on the page of the point scrolled to the top-left corner of the viewing area.

**See also** The `pageXOffset` (page 60) and `pageYOffset` (page 60) properties.

## setInterval

Method. Repeatedly calls a function or evaluates an expression.

**Syntax** When calling a function:

```
intervalID=setInterval(functionName,msec,[arg1[...],argN])
```

When evaluating an expression:

```
intervalID=setInterval("expression",msec)
```

**Method of** frame, window

**Parameters** *intervalID* is an identifier to be used only to cancel the repeated action using the `clearInterval` method.

*functionName* is the name of any function.

*msec* is a numeric value, numeric string, or property of an existing object representing millisecond units.

*arg1* through *argN* are arguments, if any, passed to *functionName*.

*expression* is a string expression or property of an existing object. The *expression* must be quoted or `setInterval` evaluates it immediately.

**Description** The `setInterval` method sets up an interval, specified in milliseconds, at which the specified function is called or specified expression is evaluated repeatedly, until the associated window or frame is destroyed or the interval is canceled by calling the `clearInterval` method.

**See also** The `clearInterval` (page 64) and `setTimeout` (page 74) methods.

## setTimeout

Method. Calls a function or evaluates an expression after a specified period of time elapses.

**Syntax** When calling a function:

```
timeoutID=setTimeout(functionName,msec,[arg1[...],argN])
```

When evaluating an expression:

```
timeoutID=setTimeout("expression",msec)
```

**Method of** frame, window

**Parameters** *timeoutID* is an identifier to be used only to cancel the impending action using the `clearTimeout` method.

*functionName* is the name of any function.

*msec* is a numeric value, numeric string, or property of an existing object representing millisecond units.

*arg1* through *argN* are arguments, if any, passed to *functionName*.

*expression* is a string expression or property of an existing object. The *expression* must be quoted or `setTimeout` evaluates it immediately.

**Description** The `setTimeout` method sets up a period, specified in milliseconds, after which the specified function is called once or specified expression is evaluated once. The action does not occur repeatedly. For repetitive timeouts, use the `setInterval` method.

**See also** The `setInterval` method (page 73) and `clearTimeout` method (see the *JavaScript Guide*).

## stop

Method. Stops the current download or current script execution; equivalent to the user pressing the Navigator Stop button.

**Syntax** [*windowReference*.]stop()

**Method of** window

## substr

Method. Returns a subset of the characters in a string.

**Syntax** `stringName.substr(start, [length])`

**Method of** string

**Parameters** *stringName* is a variable representing any string. A literal string cannot be used.

*start* is the index of the first character to extract.

*length* is optional and specifies the number of characters to extract.

**Description** The index of the first character in the source string is 0, and the index of the last character in the source string is *stringName.length*-1. If *length* is omitted, the `substr` method extracts characters from *start* to the end of the source string. If *length* is 0 or negative, no string is extracted.

## toString

Method. Converts an object or an array to a literal.

**Syntax** `identifier.toString()`

**Method of** window, document

**Parameters** *identifier* is an identifier of an object or array.

**Description** The `toString` method returns the persistent, literal form of an object or array, which is useful for debugging or as a source for another JavaScript program. An object literal has the form

```
{property1:value1, property2:value2 ...propertyN:valueN}
```

An array literal has the form

```
[element0, element1 ...elementN]
```

This behavior of the `toString` function requires the `LANGUAGE` attribute of the `<SCRIPT>` tag to be specified as `<SCRIPT LANGUAGE="JavaScript1.2">`.

**See also** The description of the `toString` method in the *JavaScript Guide*.

# Objects

This section describes the new objects and object syntax in JavaScript 1.2.

## New Objects

JavaScript 1.2 includes two new objects: the `event` object, defining an event model for the client user interface, and the `screen` object, which contains information about the client display screen.

### event

Object. Contains information about events. The event object is passed as an argument to an event handler when an event occurs.

**Syntax** `event.propertyName`

**Parameters** `propertyName` is one of the properties listed in the following table:

Property	Description
<code>type</code>	String representing the type of event. The event types are: <code>MOUSEDOWN</code> , <code>MOUSEUP</code> , <code>MOUSEOVER</code> , <code>MOUSEOUT</code> , <code>MOUSEMOVE</code> , <code>CLICK</code> , <code>DBLCLICK</code> , <code>KEYDOWN</code> , <code>KEYUP</code> , <code>KEYPRESS</code> , <code>DRAGDROP</code> , <code>FOCUS</code> , <code>BLUR</code> , <code>SELECT</code> , <code>CHANGE</code> , <code>RESET</code> , <code>SUBMIT</code> , <code>SCROLL</code> , <code>LOAD</code> , <code>UNLOAD</code> , <code>ABORT</code> , <code>IMGLOAD</code> , <code>IMGABORT</code> , <code>IMGERROR</code> , <code>MOVE</code> , <code>RESIZE</code> , and <code>HELP</code> .
<code>layerX</code>	Number specifying either mouse horizontal position in pixels, relative to the layer in which the event occurred, or object width (when passed with <code>RESIZE</code> event).
<code>layerY</code>	Number specifying either mouse vertical position in pixels, relative to the layer in which the event occurred, or object height (when passed with <code>RESIZE</code> event).
<code>pageX</code>	Number specifying mouse horizontal position in pixels, relative to the page.
<code>pageY</code>	Number specifying mouse vertical position in pixels, relative to the page.
<code>screenX</code>	Number specifying mouse horizontal position in pixels, relative to the screen.

Property	Description
<code>screenY</code>	Number specifying mouse vertical position in pixels, relative to the screen.
<code>which</code>	Number specifying either which button was pressed on the mouse or the ASCII value of a pressed key.
<code>modifiers</code>	String specifying the modifier keys associated with a mouse or key event. Modifier key values are: <code>ALT_MASK</code> , <code>CONTROL_MASK</code> , <code>SHIFT_MASK</code> , and <code>META_MASK</code> .
<code>data</code>	Varies according to event type.
<code>target</code>	String representing the object to which the event was originally sent.

## screen

Object. Contains information about the display screen resolution and colors.

**Syntax** `screen.propertyName`

**Parameters** `propertyName` is one of the properties listed in the following table:

Property	Description
<code>availHeight</code>	Specifies the height of the display screen minus any standard user-interface features displayed by the OS (such as the menu bar on Mac OS and the task bar on Windows).
<code>availWidth</code>	Specifies the width of the display screen minus any user-interface features displayed by the OS.
<code>height</code>	Specifies the total height of the display screen in pixels.
<code>width</code>	Specifies the total width of the display screen in pixels.
<code>pixelDepth</code>	Specifies the number of bits per pixel in the display.
<code>colorDepth</code>	Specifies the number of colors displayable, using the color palette if one is in use, or using the pixel depth if not.

## Using Literal Syntax

This section describes new syntax in JavaScript 1.2 that you can use to create arrays and objects.

## Creating Arrays

In addition to using constructor methods to create arrays, you can create them using literal notation.

**Syntax** `arrayName = [element0 ..., elementN]`

**Parameters** `arrayName` is the identifier of the new array.

`element0` through `elementN` compose a comma-separated list of values for the elements of the array. The array is initialized with these values, and its length is set to the number of arguments specified.

**Description** JavaScript interprets an array created using literal notation only once, when its containing HTML page is initially loaded.

## Creating Objects

In addition to using constructor methods to create objects, you can create them using literal notation.

**Syntax** `objectName = {property1:value1 ... ,propertyN:valueN}`

**Parameters** `objectName` is the identifier of the new object.

`property1:value1` through `propertyN:valueN` compose a comma-separated list of property-value pairs belonging to the object. The object is created with the number of properties specified and each property is initialized with its associated specified value.

**Description** JavaScript interprets an object created using literal notation only once, when its containing HTML page is initially loaded.

Literal object syntax can be nested. That is, the value assigned to a property can itself be an object with multiple properties, as in the following example.

**Example** The following syntax creates an object named `myCar` with three properties named `color`, `wheels`, and `engine`. The `engine` property is itself an object with two properties named `cylinders` and `size`.

```
myCar = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

# Events

This section describes new and revised events implemented in JavaScript 1.2. All event handlers now have an event object passed as an argument to the event handler. Not all properties are used for all events, as described with each event in this section.

## Click

Event. Indicates that a MouseDown event and a MouseUp event both occurred over a link or button.

<b>Handler syntax</b>	<code>onClick="handlerText"</code>
<b>Parameter</b>	<i>handlerText</i> is JavaScript code or a call to a JavaScript function.
<b>Event of</b>	link, button, radio button, checkbox, submit button, reset button
<b>Event object properties used</b>	<code>layerX</code> , <code>layerY</code> , <code>pageX</code> , <code>pageY</code> , <code>screenX</code> , <code>screenY</code> contain the coordinates of the click.  <code>which</code> contains the value 1 for the left mouse button or 3 for the right mouse button.  <code>type</code> contains the string <code>CLICK</code> .
<b>Description</b>	If the event handler returns false, the default action of the object is canceled.

## DbClick

Event. Indicates that a double mouse-click event occurred over a link or button.

<b>Handler syntax</b>	<code>onDbClick="handlerText"</code>
<b>Parameter</b>	<i>handlerText</i> is JavaScript code or a call to a JavaScript function.
<b>Event of</b>	link, button, radio button, checkbox, submit button, reset button
<b>Event object properties used</b>	<code>layerX</code> , <code>layerY</code> , <code>pageX</code> , <code>pageY</code> , <code>screenX</code> , <code>screenY</code> contain the coordinates of the click.  <code>which</code> contains the value 1 for the left mouse button or 3 for the right mouse button.

`type` contains the string `DBLCLICK`.

**Description** If the event handler returns false, the default action of the object is canceled. The interval between the mouse clicks that defines a double click event is defined by the client OS platform.

The `DbLcIck` event is not implemented on the Mac OS platform.

## DragDrop

Event. Indicates that objects have been dropped on the window.

**Handler syntax** `onDragDrop="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** window

**Event object properties used** `data` returns an array of strings containing the URLs of the dropped objects.  
`type` contains the string `DRAGDROP`.

**Description** The DragDrop event is generated when link or file objects are dropped on the window. The default action is for Navigator to load the URLs. If the event handler returns false, the default action is canceled.

**Protected** Accessing the `data` property of a DragDrop event requires signed code.

## KeyDown

Event. Indicates that a key was pressed.

**Handler syntax** `onKeyDown="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** document, text area, link, image

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the mouse was located when the KeyDown event occurred.

`which` contains the ASCII value of the key that was pressed.

`modifiers` contains the list of modifier keys held down when the KeyDown event occurred.

`type` contains the string `KEYDOWN`.

**Description** A `KeyDown` event is generated whenever the user presses a key. If the event handler returns `false`, all keypresses following the `KeyDown` event are canceled. The `KeyDown` event is sent only once when a key is pressed, even if the user holds the key down long enough to trigger automatic repetition.

## KeyPress

Event. Indicates that a key was pressed or held down.

**Handler syntax** `onKeyPress="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** document, text area, link, image

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the mouse was located when the `KeyDown` event occurred.

`which` contains the ASCII value of the key that was pressed.

`modifiers` contains the list of modifier keys held down when the `KeyPress` event occurred.

`type` contains the string `KEYPRESS`.

**Description** A `KeyPress` event is generated immediately after a `KeyDown` event, unless the `onKeyDown` handler triggered by the `KeyDown` event returns `false`. A `KeyPress` event is sent when the user initially presses a key and again for each automatic repetition generated by the keyboard.

## KeyUp

Event. Indicates that a key was released.

**Handler syntax** `onKeyUp="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** document, text area, link, image

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the mouse was located when the `KeyUp` event occurred.

which contains the ASCII value of the key that was released.

`modifiers` contains the list of modifier keys held down when the KeyUp event occurred.

`type` contains the string `KEYUP`.

**Description** A KeyUp event is generated when a key is released. This event causes no default action needing to be canceled.

## MouseDown

Event. Indicates that a mouse button was pressed.

**Handler syntax** `onMouseDown=" handlerText "`

**Parameter** *handlerText* is JavaScript code or a call to a JavaScript function.

**Event of** link, button, document

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the MouseDown event occurred.

which contains the value 1 for the left mouse button or 3 for the right mouse button.

`modifiers` contains the list of modifier keys held down when the MouseDown event occurred.

`type` contains the string `MOUSEDOWN`.

**Description** If the event handler returns false, the default action (entering drag mode, entering selection mode, or arming a link) is canceled. In that case, because the link is not armed, releasing the mouse button does not cause a Click event.

## MouseMove

Event. Indicates that the mouse was moved.

**Handler syntax** `onMouseMove=" handlerText "`

**Parameter** *handlerText* is JavaScript code or a call to a JavaScript function.

**Event of** none

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the `MouseMove` event occurred.

`type` contains the string `MOUSEMOVE`.

**Description** The `MouseMove` event is never sent unless its capture is requested by an object (see the `captureEvents` method on page 63). The `MouseMove` event cannot be canceled. If the `MouseMove` event is used to implement layer drag effects, it should be used with the `MouseDown` event.

## MouseOut

Event. Indicates that the mouse moved out of the area of the object.

**Handler syntax** `onMouseOut="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** `link`, `layer`, `area`

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the `MouseOut` event occurred.

`type` contains the string `MOUSEOUT`.

## MouseOver

Event. Indicates that the mouse moved into the area of the object.

**Handler syntax** `onMouseOver="handlerText"`

**Parameter** `handlerText` is JavaScript code or a call to a JavaScript function.

**Event of** `link`, `layer`, `area`

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the `MouseOver` event occurred.

`type` contains the string `MOUSEOVER`.

## MouseUp

Event. Indicates that a mouse button was released.

**Handler syntax** `onMouseUp=" handlerText "`

**Parameter** *handlerText* is JavaScript code or a call to a JavaScript function.

**Event of** link, button, document

**Event object properties used** `layerX`, `layerY`, `pageX`, `pageY`, `screenX`, `screenY` contain the coordinates where the MouseUp event occurred.

`button` which contains the value 1 for the left mouse button or 3 for the right mouse button.

`modifiers` contains the list of modifier keys held down when the MouseUp event occurred.

`type` contains the string MOUSEUP.

**Description** If the event handler returns false, the default action is canceled.

## Move

Event. Indicates that a window was moved.

**Handler syntax** `onMove=" handlerText "`

**Parameter** *handlerText* is JavaScript code or a call to a JavaScript function.

**Event of** window

**Event object properties used** `screenX` and `screenY` represent the new position of the top left corner of the window or frame.

`type` contains the string MOVE.

## Resize

Event. Indicates that a window's size was changed.

**Handler syntax** `onResize=" handlerText "`

**Parameter** *handlerText* is JavaScript code or a call to a JavaScript function.

**Event of** window

**Event object properties used** width and height represent the new width and height of the window, respectively, in pixels.

type contains the string `RESIZE`.

## Events

# Security Considerations

This chapter discusses aspects of the Netscape security model that are important to channel developers. It is possible that Java and JavaScript programs downloaded with channel content will need to be digitally signed in order to execute on the user's client system. This aspect of the security model is called the Netscape capabilities model.

Most channels do not need to use code that requires signing. The actions that do require signed code are listing in “Actions Requiring Signed Code”.

For more complete information about the Netscape security model and code signing, refer to the URL

<http://developer.netscape.com/library/documentation/signedobj/>

For information about using signed scripts with JavaScript 1.2, refer to the URL

[http://developer.netscape.com/library/documentation/communicator/jsguide/js1\\_2.htm](http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm)

## Netscape Capabilities Model

The distribution via networks of executable software, such as JavaScript and Java applets, introduces security concerns for users. Downloaded programs may try to perform certain actions that are potentially harmful, such as writing to the user's hard disk or removing basic interface elements from the user's control.

In previous releases of Java and JavaScript, security was handled by ensuring that downloaded programs were simply unable to perform such potentially harmful actions—they were required to execute “inside the sandbox.” The capabilities represented by certain of the new features in Java and JavaScript, however, go beyond these limitations and could be misused. Therefore, Netscape has established a capabilities model that limits those actions to be performed only under specific conditions with the end user’s permission.

## Object-Signing Protocol

The **Netscape capabilities model** is based on an object-signing protocol that enables developers to create applications that can request fine-grained access to a user’s system resources. The user then decides whether or not to grant the request.

The key capabilities of the object-signing protocol are as follows:

- allows a fine-grained continuum of access privileges (not all or nothing)
- limits access to the lines of code that need protection
- limits the duration of access to the duration of the call in which access is enabled

Java and JavaScript methods that perform potentially harmful actions cannot execute unless the code-bearing object containing the method call is signed with a digital signature. Signed objects can be Java applets, JavaScript files, plug-ins, archives, or any other kind of file.

Object signing provides users with a means of evaluating the reliability of downloaded software in much the same way they evaluate shrink-wrapped software. After users decide which software providers they trust, they install into their copy of Communicator a cryptographic certificate for each of those providers. Thereafter, Communicator uses the digital signature associated with a signed object to determine if the code is from one of the trusted providers.

## Authentication Process

When a signed object is downloaded, Communicator presents the user with a dialog box that indicates who signed the object and which capabilities it has requested. Based on that identification, the user decides whether to give the object the requested access to the computer. The code is allowed to perform only the requested actions. If the user so specifies, Communicator records the capabilities the user has assigned to the code's signer and transparently allows that signer to perform those actions in the future.

## Certificates and Digital Signatures

A **certificate** is an electronic document used to identify an individual, company, or other entity. You can use your own certificate to digitally sign a message, file, web page, and so on. A **signing certificate** is a special kind of certificate that allows you to associate your digital signature with any kind of file or JavaScript script.

Digital signatures allow the user's copy of Communicator to perform two operations important to the user:

- confirm the identity of the individual, company, or other entity whose digital signature is associated with the downloaded files
- check whether the files have been tampered with since being signed

The user (or the user's system administrator) decides which certified software providers are acceptable. The digital signature ensures that the code performing the action in question is indeed from the trusted provider.

Object-signing certificates, like other kinds of certificates, are provided by software certificate authorities such as VeriSign, Incorporated. There are several levels of object-signing certificates offering progressive degrees of assurance as to the veracity of the signer.

More information about certificates and digital signatures is available at the URL <http://developer.netscape.com/library/documentation/signedobj/>

# Using the JAR Packager

If you have code such as JavaScript scripts that must be signed, you can use the JAR Packager to sign them. The **JAR Packager** is a software tool that runs in Communicator. It compresses, envelopes, and signs any set of files, such as scripts, plug-ins, or applets into a **Java archive**.

## Java Archives

The output of the JAR Packager is a compressed collection of files arranged according to a specification called the Java archive (JAR) format. Although the files in the Java archive don't need to be signed, they can each be associated with a digital signature stored in the same archive.

Because the JAR format envelopes the compressed files, you can digitally sign an entire group of files in one step. In addition, the JAR packager need not understand the internal structure of any individual file in the archive, so the JAR packager is cross-platform and will be useful in the future for types of files not yet invented.

## Signing Scripts

To enable the JAR Packager to sign archives, you must acquire one or more signing certificates from a certificate authority. Then you must install the signing certificate you intend to use in your own copy of Communicator.

Once you have installed your signing certificate, you can use the following steps to create signed scripts:

1. Include the `ARCHIVE` and `ID` attributes in the first script's `<SCRIPT>` tag. Subsequent scripts that are included in the JAR archive can be identified by their `ID` attribute. Generally, you should use the `ARCHIVE` attribute to invoke the first script from the HTML page at your channel's top-level URL.

2. In each script, you must specifically enable your privileges before you execute any methods in secure mode (that is, with expanded privileges). After the privileged call, disable secure mode. If you call a subroutine in secure mode, it is automatically disabled when control returns to the caller's scope.
3. Compress all your scripts into a single archive and sign it with the JAR Packager.

It's also possible to sign inline scripts using a tool called Page Signer. In addition to the JAR Packager, a separate command-line tool is available called the JAR Packager Command Line. For complete documentation describing how to use the JAR Packager, see

<http://developer.netscape.com/software/signedobj/jarpack.html>

## Requesting Expanded Privileges

In addition to being digitally signed, code requiring secure capabilities must explicitly request specific privileges using a **target** that represents the system resources for which access is being requested. This feature of the Netscape capabilities model enables fine-grained access to system resources.

Requesting privileges requires only one line of JavaScript code. For example, the following method call requests access to resources named by the target *someTarget*, which represents a privilege level:

```
netscape.security.PrivilegeManager.enablePrivilege("someTarget");
```

The privilege level automatically reverts to its previous level at the end of the current scope of execution, but it is good programming practice to call the `revertPrivilege` method explicitly when you are finished using expanded privileges.

The levels of privileged access to the client system are represented by targets, which can be specified as primitive or macro targets. Primitive targets represent specific privileged actions; macro targets group primitive targets functionally. Most privileged actions of interest to channel development can be enabled using the `CanvasAccess` macro target, which enables displaying HTML text or graphics on any part of the screen, without window borders, toolbars, or

menus. The `CanvasAccess` macro target includes the primitive target `UniversalBrowserWrite`. Use the following method call to enable these privileges:

```
netscape.security.PrivilegeManager.enablePrivilege("CanvasAccess");
```

For more information about targets, see “Introduction to the Capabilities Classes” at the URL

<http://developer.netscape.com/library/documentation/signedobj/>

## Actions Requiring Signed Code

You can create a channel with compelling content, including layers displaying rich media, that does not require signed code. A digital signature is required only for code performing actions “outside the sandbox.”

Certain actions requiring capabilities to be granted are listed in this section, but the list is not exhaustive. For more information, refer to the document “Introduction to the Capabilities Classes” at the URL

<http://developer.netscape.com/library/documentation/signedobj/>

The following operations require access to the primitive target listed in parentheses:

- Writing to the hard disk (`UniversalBrowserWrite`)
- creating windows without controls or border elements (`UniversalBrowserWrite`)
- adding or removing the title bar, status bar, menu bar, location bar, personal bar, or menus (`UniversalBrowserWrite`)
- creating dependent windows (`UniversalBrowserWrite`)
- unconditional ability to close browser windows (`UniversalBrowserWrite`)
- opening a window smaller than 100 by 100 pixels (`UniversalBrowserWrite`)
- positioning a window offscreen (`UniversalBrowserWrite`)

- accessing scripts across domain boundaries, including other frames within a single frame set (`UniversalBrowserWrite`)
- setting a property on an event (`UniversalBrowserWrite`)
- getting the data from a DragDrop event (`UniversalBrowserRead`)
- submitting a form to a mailto or news URL (`UniversalSendMail`)
- getting the properties of the history object (`UniversalBrowserRead`)
- setting a file upload widget (`UniversalFileRead`)
- using the `navigator.preference` property (`UniversalBrowserRead`)
- setting the `navigator.preference` property (`UniversalBrowserWrite`)
- using an about URL other than an `about:blank` URL (`UniversalBrowserRead`)

Although Communicator writes to the hard disk when the content of any channel is cached, and it opens a chromeless window for any webtop, these actions do not require the channel code to be signed. The reason is that these actions are performed by Communicator itself under authority of Netscape's digital signature. The channel's JavaScript must be signed when it calls a method that performs one of these actions directly.

## Usage Rules for Signed Code

The following rules apply to the use of signed scripts.

- If a previously signed script is altered in any way, it must be signed again.
- For any one script to request privileges, all scripts invoked from the same HTML page must be signed.
- You cannot sign code that uses the URL syntax  
`javascript:`  
This feature cannot be used on a page with signed scripts.
- Event handlers don't include the `ARCHIVE` attribute; it must be specified in a script preceding the handler.

## Usage Rules for Signed Code

# Castanet Channels

This chapter provides information from Marimba, Incorporated, describing the Castanet channel technology that is integrated into Netcaster.

Netcaster has built-in support for two kinds of channels:

- **web-server channels**, which are described in previous chapters of this book, and
- **Castanet channels**, which are described in this chapter.

Web-server and Castanet channels are complementary. Developers should consider their user and organization needs, then implement a channel with the technology that best satisfies those needs. Castanet channels are especially appropriate for software channels, and for any channel in which scalability, personalization, robustness, or logging is a priority.

## What are Castanet Channels?

Broadly speaking, Castanet channels are similar to web-server channels: they are collections of files that, from a user's point of view, automatically install and update themselves over the network. Web-server channels are hosted on an HTTP server, while Castanet channels are hosted on a **Castanet Transmitter**.

Castanet channels offer certain benefits, which are described in the following sections.

## **Software Applications As Well As HTML**

A Castanet channel can consist of HTML pages or an entire web site: all the pages (including images, sounds, and so on) subordinate to a particular directory. But, a Castanet channel can also be a Java application or applet: an executable channel that is installed and updated automatically. Most existing Java applets and applications can be made into Castanet channels with few or no code changes.

## **Optimized Delivery of Updates**

A Castanet Transmitter (the analog of an HTTP server) knows the exact current content of each channel it serves. For any update request from a subscribed Netcaster, the Transmitter very quickly computes what needs changing in the requesting Netcaster's version of the channel: which files need to be added, deleted, or updated. In a single TCP connection, the Transmitter immediately delivers just the changes. Further, to minimize bandwidth consumption, small changes to large files are delivered as editing instructions that are executed by Netcaster. Optimized delivery makes Castanet channels well-suited to popular and fast-changing channels, reducing bandwidth usage as well as the load on the server machine.

## **User Feedback, Logging, and Personalization**

Both Java and HTML-based Castanet channels can send user feedback data back to the Transmitter in update requests. Java channels can return any sort of data; HTML channels return records of image displays and user clicks on links. By default, the Transmitter writes the feedback data to a log file which can be analyzed offline.

Alternatively, a developer can write a Transmitter plug-in that analyzes the data in real time, and optionally alters the collection of files returned to the Netcaster. It's possible, in other words, to dynamically select content for a particular user based on that user's preferences or previous interactions with the Castanet channel. The combination of feedback, logging, and plug-ins enables personalization on a subscriber-by-subscriber basis.

## Scalable Distributed Server Architecture

A Castanet-based channel can be scaled to millions of subscribers, all around the world. A Transmitter can be enhanced with multiple "repeater" Transmitters to balance the load among multiple servers and provide faster response by locating Transmitters geographically near subscribers.

## Transactional Integrity

The Internet being what it is, file transfers do not always succeed. The Castanet channel protocol maintains channel integrity in the face of network problems. As files are received from a transmitter, they are placed in a temporary holding area. Only when the final file has been received is the Castanet channel updated. There's no possibility of a user encountering a partially updated (inconsistent) Castanet channel.

## User's Perspective

Castanet channels are cleanly integrated into Netcaster, making the user interface for subscribing and launching identical to Netcaster channels. Castanet HTML channels can be displayed in a browser window or in webtop mode. Castanet application and applet channels are displayed in windows.

Users may click on a link of a Castanet channel Transmitter (or type or paste the URL into Communicator) to receive a list of the channels served by that Transmitter. Clicking on a channel name in the list subscribes to and begins execution of the channel. URLs may contain embedded commands, as described later in this chapter in Controlling Castanet Channels.

## Developer's Perspective

You can create a Castanet HTML or executable channel with the web page or Java development tools you already have. To test and demonstrate a Castanet channel, you need two Castanet software products, a transmitter and a publisher. You can download evaluation copies of both, for Win32 or Solaris, from Marimba's web site at the following URL:

<http://www.marimba.com>

### In General

Developing a Castanet channel centers on populating the channel's base directory. The channel consists of a subset of the files subordinate to the base directory. (You don't want to transmit your .java or .bak files, for example.) If you have a web site, or Java application or applet that you want to "channelize," you can probably use it as the base directory. Note that you can use Netcaster libraries in an executable Castanet channel, but such a channel will only run in the Netcaster environment.

In addition to the channel content files, a Castanet channel's base directory contains a properties file, and optionally, a plug-in subdirectory. The properties file specifies the name of the channel, its type (such as HTML or application), how often it should be updated, and so forth. You can create the properties file with a text editor or the Castanet publisher which is described shortly. The plug-in subdirectory contains the plug-in's files, if any.

Having populated the channel's base directory with files, you publish the channel with the Castanet publisher tool. The publisher has both a graphical and a command-line interface, in case you want to drive it with a script. Publishing a channel copies its files to a transmitter's channel directory: the directory it serves channels from. If multiple channels use the same file (such as a library or logo), only one copy is maintained in the transmitter channel directory. Publishing also installs the channel's plug-in, if there is one, in the transmitter and makes the channel available for subscription. Notice that it's the act of publishing that lets a transmitter know exactly what files constitute a channel.

To update a channel you repeat the procedure. You change one or more files in the base directory, and republish. The publisher copies just the new or updated files to the transmitter, and the transmitter updates its record of the

channel's current composition. The transmitter begins serving the new version of the channel to new requests, and maintains the old version for requests already in process. The transmitter automatically garbage collects obsolete files from its channel directory.

## Creating an HTML Channel

You create a Castanet HTML channel in the same way you create any web site. When you have the pages working as you want them to, you add a properties file to the web site's base directory, specifying in it the channel's name, which of its pages should be displayed first, and how often Netcaster should check for updates. Then you publish the HTML channel as described above.

All pages of a Castanet HTML channel must be subordinate to the base directory, but those pages may link to arbitrary pages outside the Castanet channel. Castanet HTML channels can use all HTML features and embed all media types. Castanet HTML channels running in Netcaster can be "instrumented" so they log user clicks on links and views images, as described in "Logging Link Clicks and Image Displays" later in this chapter.

## Creating an Application Channel

A Castanet application channel implements the Castanet interface called `Application`. Pre-built superclasses are available for application channels that display user interfaces in AWT frames, and for those that use a Castanet Bongo presentation as a user interface. (See [www.marimba.com](http://www.marimba.com) for information describing the Bongo presentation builder.) The `Application` interface is similar to the standard Java applet interface; the methods to be implemented are `setContext()`, `start()`, `stop()`, and `handleEvent()`. It's possible to create a program that works both as a conventional Java application and as a Castanet application channel.

## Creating an Applet Channel

Most existing applets can be published as Castanet applet channels without changing their code. Developers who want to create more advanced applet channels can make use of the Castanet application channel API, which provides

facilities for reading and writing files in a designated local directory, sending feedback data to the transmitter, and handling updates on the fly. Large applets launch faster when they are configured as applet channels, because they are loaded from the local disk rather than over the network; yet they retain the automatic installation and update features of ordinary applets. With the exception of being able to read and write one directory, applet channels are subject to the same security controls as ordinary applets.

## Feedback and Personalization

Although people tend to focus on the downloading aspect of channels, Castanet channels can also upload data, called feedback data, to their transmitters. The transmitter automatically logs the data for offline analysis. Developers can also gain real time access to the feedback data by writing a piece of code called a transmitter plug-in. When a transmitter receives an update request from a Castanet channel instance, it invokes the channel's plug-in. The plug-in receives the feedback data and the list of files to be returned to the requesting Netcaster. The plug-in can respond to the feedback data as its developer sees fit. A simple plug-in might just record the data in a different format than the transmitter does by default. A more complex plug-in can use the feedback data to personalize the channel instance by altering the list of files destined for the Netcaster. For example, if the feedback data specifies that the user is French, the plug-in can add French-language files. If the feedback data indicates that the user has clicked on a particular advertisement, the plug-in can return a related ad. Or, if the feedback data contains a user ID or account number, the plug-in can look up user-specific data in a database.

A plug-in is realized as a Java subclass of the transmitter's default plug-in which logs user feedback data. A plug-in can be created with any Java development tools and can contain native methods. Its files are stored in the channel's base directory. Publishing the channel copies the plug-in executable to the transmitter and installs it; to update the plug-in, you just republish the channel.

# Castanet Channel Extensions for Netcaster

Marimba has extended the capabilities of Castanet channels that run in Netcaster. Developers of HTML channels can use these extensions to let users control Castanet channels from web pages, and to log the images shown to a user and the links the user clicks on.

## Controlling Castanet Channels

Castanet channels may be addressed by these kinds of URLs:

1. `http://host [ :port ] /channelname`
2. `castanet://host [ :port ] /channelname [ ?command ]`

Use the `http:` form in a web page that you want to work in both Netcaster and non-Netcaster browsers.

A URL of the `castanet:` form can be interpreted directly by the browser and can be used when the user is offline. The `http:` version of the URL works only when the user is currently online since it requires access to the transmitter.

Here is an example of a URL to start the channel Binary Tree on the transmitter `trans.marimba.com`:

```
<A HREF="castanet://trans.marimba.com/Binary_Tree">
Binary Tree
</A>
```

A `channelname` value may consist of any combination of the alphabetic characters (`a-z` and `A-Z`), numerals (`0-9`) and the underscore (`_`) character. If the actual channel name includes any other characters, such as space, substitute an underscore for each such character. For example, if a Castanet channel has been published with the name “The \*Special\* Channel!” use this name in a URL:

```
The__Special__Channel_
```

Netcaster interprets the optional `command` values as follows:

```
subscribe
```

Subscribe to the channel (download it).

`start`

Launch a channel, subscribing to it first if necessary.

`stop`

Stop a running channel (no effect on HTML channels).

`update`

Send the channel's transmitter an update request.

`log=[ tag, ]URL`

Add an entry containing the tag or URL value to the channel's log file (as described in "Logging Link Clicks and Image Displays").

Here's an example of a URL with a command. If a Netcaster user clicks on the following link:

```
<A HREF="castanet://trans.marimba.com/Binary_Tree?start">
Launch Binary Tree Channel
</A>
```

Netcaster will launch the channel called `Binary_Tree`, downloading it from `trans.marimba.com` if necessary. Typing or pasting the URL into Communicator produces the same response.

## Logging Link Clicks and Image Displays

Cross-reference and image tags in Castanet HTML channels may contain extra information for logging user clicks and image displays. Here's an example that shows how an ad can be logged:

```
<A HREF="?log=http://www.example.com/index.html">
<IMG SRC="?log=ExampleAd.gif">
</A>
```

When a user displays the page containing this link, Netcaster will add the string `ExampleAd.gif` to the Castanet channel's local log file. If the user clicks on the ad, Netcaster will add `http://www.example.com/index.html` to the log. The logging is invisible to the user, to whom the link and the image behave normally.

When Netcaster makes an update request to the transmitter, it sends the log entries in the request. The transmitter passes the log entries to the Castanet channel's plug-in, if there is one, otherwise it appends the entries to a log file that holds data from all instances of the channel served by that transmitter. Netcaster clears a Castanet channel's log file when it receives a channel update from a transmitter.

You can direct Netcaster to substitute a string of your choosing for the URL that's written to the log by default. For example:

```
<A HREF="?log=ClickedOnIndex,http://www.example.com/index.html">
<IMG SRC="?log=GreenAdDisplayed,ExampleAd.gif">
</A>
```

These examples direct Netcaster to log `ClickedOnIndex` rather than `http://www.example.com/index.html`, and `GreenAdDisplayed` rather than `ExampleAd.gif`.

## Castanet Features and Netcaster

Castanet channels that run in Netcaster can use all Castanet facilities, with the following provisos:

- Castanet channels that use Bongo presentations must contain the Bongo libraries (`bongo.zip`) because they aren't part of Netcaster.
- Netcaster doesn't provide an `etc` folder for loading native code, such as ShockWave.

## For More Information

For complete information on developing Castanet channels, see the Castanet developer documentation at the URL:

[http://www.marimba.com/doc/Castanet\\_Developer\\_Docs/current/index.html](http://www.marimba.com/doc/Castanet_Developer_Docs/current/index.html)

To download a demonstration transmitter and publisher, or a free tuner, see Marimba's home page:

<http://www.marimba.com>

For More Information

# Glossary

<b>cache</b>	A persistent data storage area, typically located on a local or client computer system. Data that is likely to be reused can be stored (cached) locally to avoid repeated downloading.
<b>Castanet channel</b>	A channel that uses push technology designed and marketed by Marimba, Inc., to deliver web content to subscribers.
<b>Castanet transmitter</b>	The web server component of a Castanet channel.
<b>Castanet tuner</b>	The web client component of a Castanet channel.
<b>certificate</b>	An electronic document that uses public-key cryptography and the verifiable authority of a certifying entity to identify an individual, company, or other entity. See also <b>signing certificate</b> .
<b>channel</b>	A web site employing push technology to deliver content to the client without requiring immediate user interaction.
<b>Channel Finder</b>	An element of the Netcaster user interface that provides access to channels to which the user can subscribe.
<b>container</b>	A grouping of conceptually related pieces of web content, displayed within a contiguous area of the Communicator user interface.
<b>crawling</b>	The action performed by Netcaster when it periodically downloads and caches the contents of a channel.

<b>crossware</b>	Software applications that run across networks and operating systems and are based entirely on open Internet standards like HTML, Java, and JavaScript.
<b>JAR</b>	See <b>Java archive</b> .
<b>JAR Packager</b>	A software tool that compresses, envelopes, and digitally signs a set of files into a Java archive (JAR).
<b>Java archive</b>	A compressed collection of files arranged according to a specification called the Java archive (JAR) format.
<b>Netcaster</b>	The component of Netscape Communicator that enables users to receive content from channels.
<b>netcasting</b>	The delivery of web content to subscribers using push technology.
<b>netcasting parameters</b>	Configuration values maintained by Netcaster for each subscribed channel to control crawling the channel site. See also <b>crawling</b> .
<b>Netscape capabilities model</b>	An object signing protocol that enables developers to create applications that can request fine-grained access to a user's system resources.
<b>offline site</b>	A web site that can be downloaded in the background and browsed by the client without an active web connection.
<b>public key cryptography</b>	A method of encoding and decoding data that involves a freely distributed public key and a secret private key. Data encoded with either of the keys requires the other to decode it.
<b>push technology</b>	The capability of web components to deliver information from servers to clients without explicit actions of the client.
<b>robots control</b>	A protocol by which you can specify URLs on your web site that are not to be downloaded to the client by a crawler.
<b>signing certificate</b>	A special kind of certificate that allows you to associate your digital signature with any kind of file or JavaScript script. See also <b>certificate</b> .
<b>subscribe</b>	The action of a client to create a connection with a channel. The subscription process sets up parameters specifying the period and frequency of automatic downloads of web content.
<b>target</b>	A programmatic identifier representing a set of system resources to which applications can request privileged access. See also <b>Netscape capabilities model</b> .

**web-server  
channel**

A channel that uses HTTP (hypertext transfer protocol) server technology.

**webtop**

A type of channel displayed in an unframed window that can cover the entire desktop and be locked to the backmost layer of the display. Webtops display dynamic information in a static format.



# Index

## A

- absoluteTime property 29
- activate method 28
- active property 27
- addChannel method 28
- aliases 38
- animation 50
- API (application programming interface) 25
- ARCHIVE attribute 90
- authentication 89

## B

- back method 63

## C

- cache 37, 44
  - size 33
- canvas mode 48
- captureEvents method 63
- cardURL property 30
- Castanet 95
- Castanet Transmitter 95
- certificate 89
- channel
  - adding 26
  - applet 99
  - application 99
  - Castanet 95
  - controlling 101
  - defined 13
  - design 19

- elements 17
- logging 102
- object 26
- properties 27, 29–36
- sample 47
- web-server 95
- charCodeAt method 64
- clearInterval method 64
- Click event 79
- compatibility 54
- components array 25
- componentVersion property 28
- content 15
- crawling 37, 44

## D

- DbClick event 79
- debugging 42
- delete operator 55
- depth property 31
- desc property 31
- design 19
- digital signature 89
- directories property 61
- disableExternalCapture method 64
- documentation 50, 87
- download
  - depth 31
  - interval 32
  - time 29
- downloading 37–45
- drag-and-drop 49

DragDrop event 80

## **E**

enableExternalCapture method 65

equality operators 55

estCacheSize property 31

event handler 79

event object 76

events 79–85

examples 36, 47–50

## **F**

feedback 96, 100

find method 65

floating palettes 47

forward method 66

fromCharCode method 66

full-screen window 48

## **G**

getChannelObject method 29

getSelection method 66

## **H**

handleEvent method 66

heightHint property 32

home method 67

HTML 14, 18, 47–50, 96, 99

HTTP server 95

## **I**

ID attribute 90

import statement 26

information types 16

innerHeight property 57

innerWidth property 58

intervalTime property 32

## **J**

JAR Packager 90

Java 88, 96

Java applets 39

Java archive (JAR) 90

JavaScript 14, 25–37, 47–50, 53–85,  
88

documentation 53

versions 54

## **K**

KeyDown event 80

KeyPress event 81

KeyUp event 81

## **L**

LANGUAGE attribute 54

language property 56

layout 49

leftHint property 33

locationbar property 58

logging 96, 100

## **M**

maxCacheSize property 33

menubar property 58

methods 28, 63–75

importing 26

mode property 33

MouseDown event 82

MouseMove event 82

MouseOut event 83

MouseOver event 83

MouseDown event 83  
Move event 84  
moveBy method 67  
moveTo method 67

## **N**

name property 28, 34  
navigator object 56  
netcaster component 25  
netcasting parameters 51

## **O**

object signing 88  
objects 76–78  
    channel 26  
open method 68  
operators 55  
outerHeight property 59  
outerWidth property 59

## **P**

pageXOffset property 60  
pageYOffset property 60  
personalbar property 60  
platform property 57  
print method 70  
privileges 91  
properties 27, 29–36, 56–63, 79  
push technology 13

## **R**

redirection 38  
releaseEvents method 71  
Resize event 84  
resizeBy method 71

resizeTo method 71  
robots control 40  
routeEvent method 72

## **S**

sample code 36  
screen object 77  
screenX property 61  
screenY property 61  
SCRIPT tag 54  
scrollbars property 62  
scrollBy method 72  
scrollTo method 72  
security 87  
setInterval method 73  
setTimeout method 74  
signed code 87, 92  
signing scripts 90  
statusbar property 62  
stop method 74  
streamed content 39  
subscription process 26  
substr method 75

## **T**

target 91  
toolbar property 62  
topHint property 35  
toString method 75

## **U**

url property 35  
URLs  
    implicit 38  
user interface 16, 18, 33

## **W**

webtop 15, 18, 33, 47  
widthHint property 35  
window object 57

## **Z**

z-order 47