


Welcome to the InDesign/InCopy CS2 Plug-in Development training sessions.

Copyright 2005 Adobe Systems Incorporated. All rights reserved.

The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.


Adobe, Adobe InCopy, and Adobe InDesign are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks, and Mac OS is a trademark of Apple Computer, Inc. Microsoft, Windows, Windows 95, Windows 98, Windows NT and Windows XP are registered trademarks of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.



Objectives

- **Introduce InDesign/InCopy plug-in developers to the world of scripting**
- **Highlight the scripting architecture**
- **Understand responsibilities of a scriptable plug-in**
- **Introduce common development scenarios**
 - Scripting
 - INX support
 - InDesign Server support
 - Other use cases


2005 Adobe Systems Incorporated. All Rights Reserved. 2



In this session, we will start by introducing the concept of driving InDesign and InCopy using a script, and what it means to plug-in developers to support the use of plug-in features through a script. We will also discuss why this topic is important in InDesign/InCopy CS2.

Then we will discuss the application's scripting architecture at a high-level to establish a frame of reference for implementing scriptable plug-ins.


Then we will discuss what it takes to add scripting support to a plug-in. In doing so, you will gain an understanding of what responsibilities a scriptable plug-in must take on, and how to implement common scenarios when adding scriptability to a plug-in.



Helpful knowledge...

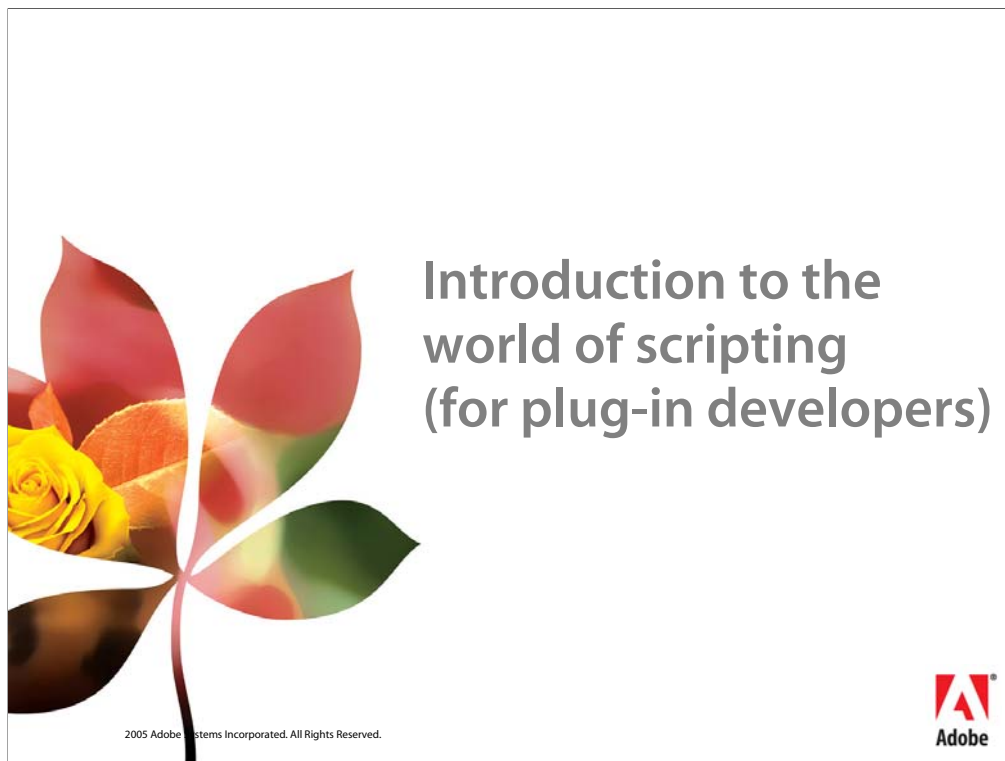
- **It would greatly help if you know how to program in one of these three languages:**
 - AppleScript
 - Visual Basic, VBScript, or any language that can be a COM client
 - Java Script


2005 Adobe Systems Incorporated. All Rights Reserved.



It is also helpful if you are familiar with any of the following languages/technologies:

- AppleScript: How to write an AppleScript
- Visual Basic, VBScript, and other COM object client languages: How to write code that uses a COM object
- JavaScript: How to write a JavaScript, especially for use with any Adobe application that includes the Extend Script engine.






What is a script?

- **A program written to automate application features**
 - Not in the form of a plug-in
 - Easier to write than a plug-in...
- **Supported programming languages**
 - Mac: Apple Script
 - Win: Any language that can be a client of a COM object
 - E.g. Visual Basic, Visual C#, etc.
 - Both platforms: JavaScript
 - Extend Script engine built into the application


2005 Adobe Systems Incorporated. All Rights Reserved. 5



In InDesign/InCopy terminology, a script is a program that uses the InDesign or InCopy scripting library to automate the features of the said applications. Usually, scripts are written in the form of a single file that contains various command calls to the application.

You can use the following languages to write a script:


- (Mac only) AppleScript
- (Windows only) Any language that can be a client of a COM object, such as Microsoft Visual Basic, Microsoft Visual C#, Microsoft Visual C++, Borland Delphi (Object Pascal), etc. You can also use the Windows Scripting Host to write VBScripts.
- (Both platforms): Java Script, using the Extend Script engine hosted in the application.



How do I learn how to write a script?

- **Script samples and reference documentation**
 - See InDesign CS2/InCopy CS2 Product CD-ROM
- **How to write a script**
 - See InDesign CS2 Scripting Guide
- **Also visit the “InDesign Scripting User-to-user forum”**
 - <http://www.adobeforums.com>

2005 Adobe Systems Incorporated. All Rights Reserved. 6




Scripts are very simple to read, as they are usually written in a top-down manner. That is, you start executing from the first line of code and work your way downwards.

If you want to see samples of scripts, the best place to look for them is the InDesign CS2/InCopy CS2 product CD-ROMs (not the debug build, but the release/retail build CD-ROM). In the CD is a folder that contains:

- Sample scripts and a read me file
- Scripting Guide that teaches you how to write scripts, and provides a reference of the scripting library in its original form (without any 3rd party scripting plug-ins added in)

You can run the scripts in the IDEs specific for each scripting language, or you can run single-file scripts that are stored in your {ApplicationFolder}/Presets/Scripts folder from the “Scripts” panel in the application. (The Scripts panel will recurse subdirectories.)


Since we are doing plug-in development training, we won’t spend a lot of time discussing how to write scripts, but if you are interested in learning more about scripting and want help beyond what is included in the product CD-ROMs, you can refer to the InDesign Scripting User-to-user forum at <http://www.adobeforums.com>.




When is a script useful?

- **When you have a repetitive user task you just want to automate quickly**
 - Quite empowering for power users of InDesign/InCopy
- **When you have a procedure that can be automated without developing a plug-in**
 - Several 3rd party productivity tools for InDesign are developed using this method

2005 Adobe Systems Incorporated. All Rights Reserved. 7




Scripting is useful for InDesign/InCopy users who want to automate lengthy tasks that must be repeated. By storing the sequence of operations into a script, you can reliably “play it back” over and over again. As a developer, you can use scripting to automate a procedure without having to develop a plug-in.



Why is scripting important for plug-in developers?

- **Allow users to automate your plug-in's features**
- **If your plug-in adds persistent data, adding scripting capabilities will add support for:**
 - Import/Export of INX-related files (InDesign Interchange format, InDesign "Snippets")
 - Library Assets
 - Package for GoLive CS2
 - Other features that depend on the INX format
- **This may be the only way to expose your plug-in's features in InDesign Server (which has no UI)**

2005 Adobe Systems Incorporated. All Rights Reserved. 8




However, you may be thinking "scripting seems like child's play – you can't do things like you can do in plug-ins – why should I invest any time in providing an alternate way to control my plug-in when I have nice menus/dialogs/panels?" As a developer, you want to consider scripting for the following reasons:

- If your plug-in adds unique features, adding "scriptability" to your plug-in allows your users to automate your plug-ins' features with scripts.
- If your plug-in adds custom persistent objects (bosses) or persistent interfaces (add-ins), adding "scriptability" to your plug-in will automatically provide support during:
 - Importing/Exporting a document in the InDesign interchange format, or parts of a document in InDesign Snippets
 - Library Assets, which in CS2 are based on INX.
 - Other features in the application that are based on the INX format.
- Scripting support is important if your plug-ins will be targeting InDesign Server, which does not have a user interface. There are certain types of features which don't require a scriptable plug-in (e.g. a feature that is extensible by service providers), however, for the most part, you will need to consider adding scripting support for your plug-in.
- (Not shown on slide) You can also write a script to rapidly prototype of a feature for your plug-in.
 - Since we are focused on automating existing features, this might be a good way to do a proof of concept.
 - The InDesign/InCopy scripting library provides rudimentary UI objects as well.
- (Not shown on slide) There are some application features that are only controllable by a script, not by the C++ API (e.g. The InDesign Package feature, whose command class ID is not provided in the API headers.)




- This corresponds to the "Scripting Architecture" section in the [Making Your Plug-in Scriptable] tech note.



Script Manager


- A service provider (kScriptManagerService) that provides basic information for supporting the scripting language
- One for each scripting “context”
 - AppleScript
 - COM
 - JavaScript
 - Others (INX, scripting tag, etc.)

2005 Adobe Systems Incorporated. All Rights Reserved. 10



One of the fundamental components in the scripting architecture the scripting manager, and it is the general engine that makes scriptable plug-ins “scriptable”.


- Script managers receive system messages (via AETE, COM, or Extend Script engine) and uses the script request handler to send the message to the appropriate script provider so that the message can be processed. Script managers are identified by IScriptManager, and there are 3 kinds that are of interest to developers who write scripts (in a later slide, we’ll discuss how to query for one of these script manager bosses):
 - kAppleScriptMgrBoss: For AppleScript support (Macintosh only)
 - kJavaScriptMgrBoss: For JavaScript support
 - kOLEAutomationMgrBoss: For COM support (Windows only)



Scripting DOM

- **The model of a document with a set of objects available to scripting clients**
 - Published by each script manager
 - Script object
 - An object in the Scripting DOM, that has events (methods) and properties
 - A script object may inherit from another
 - See interface IScript
 - Script provider
 - A boss that responds to requests made on a script object and accesses other bosses in the document or application
 - See interface IScriptProvider

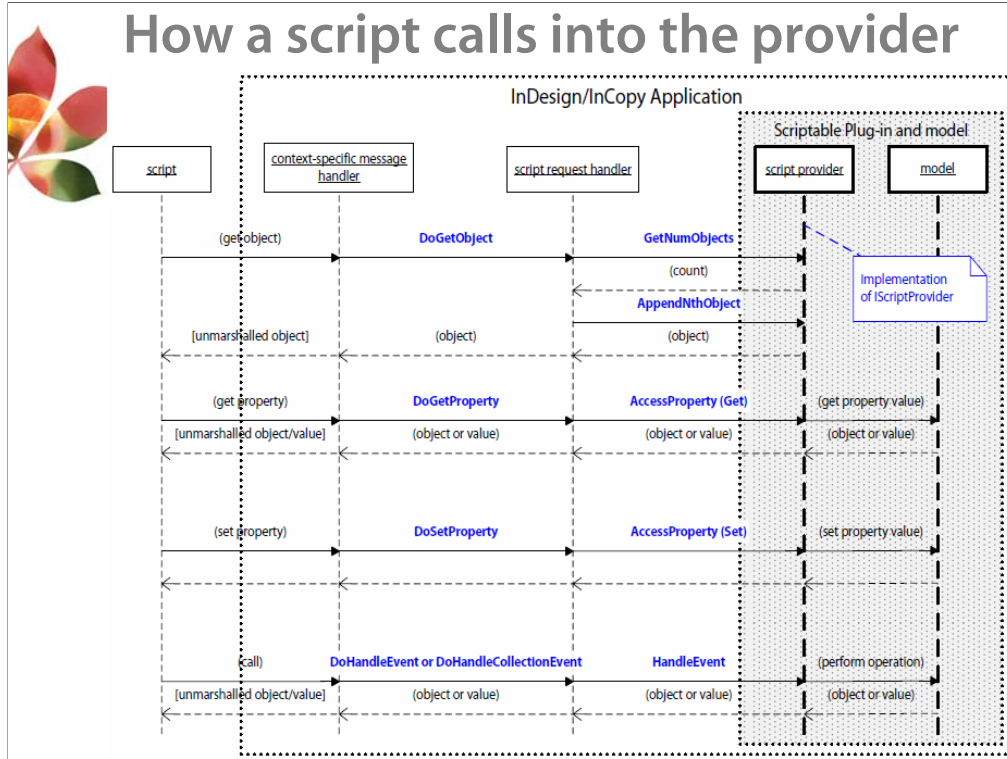
2005 Adobe Systems Incorporated. All Rights Reserved. 11



The other fundamental component of the scripting architecture are components that make up the Scripting DOM.

The Scripting DOM is the model of a document with a set of objects that are available to scripting clients. It is published per scripting context by each of the scripting manager, and it is comprised of the objects that are implemented in a scriptable plug-in:

- Script objects are objects that want to be exposed to the scripting architecture. All scriptable bosses must aggregate some implementation of IScript.
- Script providers are implementations of IScriptProvider, which provides assistance to the script manager (below) in determining how to handle a call from an external script. The call from the script can be property access or manipulation, or an event (method). The script providers declare their support by specifying additions to the Scripting DOM in the form of script objects, properties and events in a VersionedScriptElementInfo resource.



Scripts call into the application by means of sending a “message” into the application. The message can take be any of the following:

- Getting an object
- Getting or setting a property
- Calling an event (method)
- (Note: objects that can exist in a collection are created and deleted by special events)


This is generally handled by the platform specific system architecture (AETE for Macintosh, COM for Windows, Extend Script for JavaScript). The context-specific message handler receives that message, and delegates to a script request handler to dispatch the message to the appropriate script provider (that can handle the ScriptID associated with the message).

Once the script provider gets the call, it is responsible for doing one of the following:

- Get object: If the script provider represents an object, then the script provider is responsible for specifying how many instances of that object exist, and for returning references to those objects.
 - The GetNumObjects and AppendNthObject protected methods are called in the script provider implementation.
- Get property: The script provider returns the value of the requested property (hint: IScriptEventData::IsPropertyGet returns kTrue)
 - Determine what property is to be returned, and create an instance of a ScriptData class, set the value by calling the appropriate Set*** method (based on the data type declared in the VersionedScriptElementInfo resource for this property), and return the requested data via IScriptEventData::SetReturnData. (This done in the implementation of IScriptProvider::AccessProperty method.)
- Set property: The script provider sets the value of the specified property. (hint: IScriptEventData::IsPropertyPut returns kTrue)
 - Determine what property is to be set, get the ScriptData class by calling IScriptEventData::ExtractEventData, and based on the data type you expect (as declared in the VersionedScriptElementInfo resource for this property), call the appropriate ScriptData::Get*** method, and do whatever operations necessary to store the data. This could be require processing a command, or caching the value somewhere. (This done in the implementation of IScriptProvider::AccessProperty method.)
- Event: The script provider performs some operations based on input parameters, and returns necessary return values as well as output parameters.
 - You retrieve the ScriptData class in the same way you do in the “Set Property” case, and perform whatever operations necessary. You then return data to the script in the same way that the ScriptData class is returned in the “Get Property” case above. You can also return data via the parameter list by using the IScriptEventData::InsertEventData method and identifying the parameter by its ScriptID.

The API class that encapsulates the parameter in a property access or event message is ScriptData. By calling any of its “Set” methods, you are also selecting the type of data it represents.






Before you begin...

- **Determine how to represent your plug-in features in the scripting DOM**
 - Objects
 - Events (on which objects?)
 - Properties (on which objects?)
 - Enums (for which properties/parameters?)
 - Suites (AppleScript only)
- **Refactor your code so that it is easy to call from a script provider**
 - Model/View separation is strongly recommended
 - Use of adapter/helper classes recommended

2005 Adobe Systems Incorporated. All Rights Reserved. 14



One of the main scripting-related tasks you will complete as a plug-in developer interested in scripting support is to provide additional components to your plug-in so that it can participate in the scripting architecture. However, before you can make your plug-in “scriptable”, there are a few things to consider.

How do you want to represent your plug-in features in the scripting library, or in other words, what items do you need to add?

- Objects? Usually, each object you expose to the scripting architecture is represented by its own script boss.
- Events (on which objects?)
- Properties (on which objects?)
- Enums (for which properties or event parameters?)
- Suites


Another related decision is to figure out which script provider implementations support which object/event/property. If you are introducing a custom script object, you are “representing” that script object, so typically, you would provide a script provider for that object. If you are only adding events or properties, you can put all of the code into a single script provider, or you can implement multiple script providers (1 for each object you are adding events and/or properties).

Can your plug-in features be called from another component?

- Often times as plug-in developers, we find ourselves putting the bulk of our implementation code within an implementation of `IActionComponent` or `IDialogController`, since that’s the easiest place to put our model access or manipulation code. However, doing so closely ties the model and view implementations, and you will have to refactor your code if you want to add a script provider. One way to refactor your code is to put the bulk of your model access or manipulation code in helper classes, thus treating interfaces like `IScriptProvider` as an adapter to the scripting architecture. Another way to refactor your code is to implement selection `ASB/CSBs`, and as needed, implement custom commands to do model manipulation. If you do this cleanly, you can even put the script provider component into an entirely separate plug-in, allowing your users to download the script provider plug-in only when desired.
- Model/View separation is strongly recommended not only to make scripting support easier, but is required if you want to allow your plug-in to be used in InDesign Server.

See also:


- Section “Before you begin” in the [Making Your Plug-in Scriptable] tech note.




Overview of responsibilities

- 1. Register Script ID/Name pairs, and declare ScriptIDs and ScriptElementIDs in a header file
- 2. Write VersionedScriptElementInfo resources
- 3. Implement IScript for any Object(s) in your VersionedScriptElementInfo resource
- 4. Implement script provider boss(es) to support the Provider elements in VersionedScriptElementInfo
- 5. Add boss/implementation IDs for any IScript and IScriptProvider implementations

2005 Adobe Systems Incorporated. All Rights Reserved. 15




- 1. First, you must register ScriptID/name pairs at the ASN Script ID/Name pair registration page (more on the next slide). Once you have successfully registered and reserved your Script ID/name pairs, declare them along with object GUIDs in a *ScriptingDefs.h file, and declare the ScriptElementIDs using the DECLARE_PMIID(kScriptInfoIDSpace, ...) macro in your *ID.h file.
- 2. Write VersionedScriptElementInfo resources in a *.fr file for the elements you wish to add to the scripting DOM. You must have a "Provider" element. The elements you can add to the Scripting DOM include:
 - Objects: if exposing a script object
 - Enums: if adding new enums
 - Events: if adding events to existing or new script objects
 - Properties: if adding properties to existing or new script objects
 - Suites (AppleScript)



ScriptID/Name Pairs

- **4-letter IDs to identify:**
 - Object
 - Event
 - Properties and Event parameters
 - Enums/Enumerations
 - AppleScript Suites
- **Paired up with a “name”**
 - Script ID/Name pairs must NOT clash in the application
 - Register them at http://partners.adobe.com/public/developer/indesign/topic_prefix_reg.html



2005 Adobe Systems Incorporated. All Rights Reserved. 16

The ScriptID class is used extensively in script providers, as it is the main way for the script manager in a specific context to uniquely identify one of the following scripting constructs:

- Object (or class): the symbol #define for the ID is prefixed with “c_”
- Event: the symbol #define for the ID is prefixed with “e_”
- Properties and event parameters: the symbol #define for the ID is prefixed with “p_”
- Enumerations: the symbol #define for the ID is prefixed with “en_”
- AppleScript Suites: the symbol #define for the ID is prefixed with “s_”


ScriptIDs are used by script managers to determine which script provider supports the associated element, and by script provider code to distinguish the various elements (as listed above) in the scripting DOM.

The ScriptIDs used in the scripting library for the application are listed in the {SDK}/source/public/includes/ScriptingDefs.h header file. They are all enclosed in single-quotes, which makes the C++ compiler treat them as the numeric literal value indicated by their ASCII codes. There is a use case slide later in this presentation that shows you how to convert this ScriptID to a string (so you don't have to deal with big- vs. little-endian byte arrangements).

A ScriptID is paired up with a name in the VersionedScriptElementInfo resource. To help the script managers distinguish different script DOM elements, there are a set of rules regarding the use of ScriptID/name pairs in the application. The general rule of thumb is that ScriptID/Name pairs must not clash in the application. (As long as you follow this rule, you will be guaranteed to be free of clashes.) For instance, if there is an Object that has the ScriptID = 'abcd' and the Name = “abcd”, there must be no other object, event, property, enumeration, or suite that has the same ScriptID but a different Name, or the same Name but a different ScriptID. To help guarantee that this rule is followed by Adobe engineers as well as third party plug-in developers, we have set up a Script ID/Name pair registration page on the Adobe Solutions Network web site, at which you can register the ScriptID/Name pairs you will be using in your scriptable plug-in. The URL is shown on the slide.


See also:

- Section "ScriptID/Name pair registration" in the [Making Your Plug-in Scriptable] tech note.



CLSID

- **A globally unique ID to identify objects**
 - Also called GUID or UUID
- **How to generate a CLSID**
 - Windows
 - Use GUIDGEN.exe
 - Macintosh
 - Use UUID Factory or UUID Generator (see notes)
 - ...or write a small applet that calls CoreFoundation API CFUUIDCreate

2005 Adobe Systems Incorporated. All Rights Reserved. 17


For Objects, in addition to In specifying a ScriptID (singular or plural) in your VersionedScriptElementInfo resource, you also need to specify a globally unique CLSID (also known as GUID or UUID). These IDs are used primarily to help the Windows COM architecture distinguish the COM objects being operated upon. Although this is used only on the Windows platform, we recommend that you specify CLSIDs on the Macintosh platform as well, since the VersionedScriptElementInfo resource requires it for the Object element, and the CLSID of each object is compared at application startup to ensure there are no conflicts.


On Windows, a CLSID can be generated by using GUIDGEN.exe, which comes with Visual Studio .NET 2003 (see Visual Studio .NET 2003 documentation or <http://msdn.microsoft.com> for its usage).

On the Macintosh, unless you write your own applet that calls the Core Foundation library API CFUUIDCreate, there isn't an equivalent tool provided by CodeWarrior or the Apple Developer Tools CD (see <http://www.adobeforums.com/cgi-bin/webx?13@674.lotRdnLNbw2.706956@.3bb4bbc8/0>). Instead, you can download one of these tools available on the web:

- UUID Factory: http://www.apple.com/downloads/macosx/development_tools/uuidfactory.html
- UUID Generator: <http://www.versiontracker.com/dyn/moreinfo/macosx/11848>

See also:


- Section "ScriptElementIDs, ScriptIDs, Names, Descriptions and GUIDs" in the [Making Your Plug-in Scriptable] tech note.



ScriptElementID

- **An ID declared in the kScriptInfoIDSpace**
 - DECLARE_P MID(kScriptInfoIDSpace, ..., ...)
- **Used to identify an element in the VersionedScriptElementInfo resource**
- **Each ScriptElementID is associated with a ScriptID**
 - If the object identified by an ScriptElementID is a plural object, it will also be associated with the plural ScriptID

2005 Adobe Systems Incorporated. All Rights Reserved. 18




The ScriptElementID is an ID declared in the kScriptInfoIDSpace using the DECLARE_P MID macro. This ID, which must also be unique within the application, is used internally within the Scripting DOM to identify an element in the VersionedScriptElementInfo resource. It is also used when specifying inheritance for the Object element, as well as specifying which elements are to be supported by a particular script provider in the Provider element.

Each ScriptElementID is usually associated with a single ScriptID, unless the ScriptElementID refers to a plural object.


See also:

- Section "ScriptElementIDs, ScriptIDs, Names, Descriptions and GUIDs" in the [Making Your Plug-in Scriptable] tech note.



VersionedScriptElementInfo

```
resource VersionedScriptElementInfo(1)
{
  { // Contexts
    kFiredrakeScriptVersion, kCoreScriptManagerBoss, kWildFS, k_Wild,
  }
  { // Elements
    Object
    {
      kBPIPrefObjectScriptElement,           // ElementID
      c_BPIPref,                             // ScriptID
      "basicpersistinterface preference"     // Name
      "BasicPersistInterface preference",    // Description
      kBPIPref_CLSID,                       // CLSID
      NoPluralInfo,                         // plural info
      kPreferencesObjectScriptElement,      // base script object
      kPreferencesSuiteScriptElement,      // suite
    }
  }
  // . . . Provider element connects elements with the Scripting DOM
}
```



2005 Adobe Systems Incorporated. All Rights Reserved. 19

Here's a sample of a VersionedScriptElementInfo resource, taken from the BasicPersistInterface sample.

The first line specifies the resource type, VersionedScriptElementInfo, along with the resource ID of 1. You can have multiple VersionedScriptElementInfo resources in a single plug-in, so this resource ID allows you to distinguish resources of the same type.

The first group, within the curly brackets, specifies the list of contexts that are to be associated with this resource. This sample specifies that this VersionedScriptElementInfo was first introduced in 4.0 ("Firedrake"), is applicable to all script managers (core), all product feature sets, and UI locales. You could add multiple lines in this part, say, if this VersionedScriptElementInfo resource was to be targeted only for the INX script manager, only under the InDesign product feature set (not InCopy or InDesign Server), and only with the Japanese UI version.

The next group, specifies the list of elements in this VersionedScriptElementInfo resource. In this group, you can specify any number of elements, which include Objects, Suites, Events, Properties, Enums, and Providers. In this sample, we see an Object element, which has the shown attributes. More elements are listed after the close-curly-bracket. Of which, the most important one is the Provider element, which makes the connections between the elements listed in this VersionedScriptElementInfo resource and the rest of the Scripting DOM. It can set up an Object declared here as a child of another object, or it can specify that the Events and Properties declared here be added to other Objects.


This VersionedScriptElementInfo resource specifies your plug-ins contributions to the scripting DOM. Instead of introducing the entire specification of this resource at this point of the presentation, we will refer you to specific instances used in the sample plug-ins in the upcoming slides, so you have a better understanding of how to write the resource for different purposes. If you would like to look at an ODFRez type definition of the VersionedScriptElementInfo resource, see [\[SDK\]/source/public/includes/ScriptInfoTypes.fh](#).

Q: What happened to the ScriptElementInfo resource from CS?

- A: It still exists in the CS2 API for backwards compatibility with CS plug-in code, however, if you have CS plug-ins that were already made scriptable, we strongly recommend that you change your resource code to use the VersionedScriptElementInfo resource instead, for the following reasons:
 - The VersionedScriptElementInfo resource allows you to target multiple product feature set (e.g. InDesign, InCopy, InDesign Server) in one statement, while the ScriptElementInfo only allows you to target a single product feature set.
 - The VersionedScriptElementInfo resource allows you to target specific, multiple initial application versions (e.g. 3.0, 4.0) in one statement, while the ScriptElementInfo is targeted for use with script elements that were introduced at version 3.0 (InDesign/InCopy CS). This is very important if your script element info needs to be application-version specific. (Refer to the [Making Your Plug-in Scriptable] tech note for information on Versioning your script provider.)


See also:

- Section "VersionedScriptElementInfo resource" in the [Making Your Plug-in Scriptable] tech note.



Base objects in the Scripting DOM

<pre><<script-object>> object [Description = Any object, ElementID = kAnyObjectScriptElement, Plural ScriptID = c_Objects = 'OBSJ', Plural name = objects, ScriptID = c_Object = 'Obj']</pre>
<pre>+index : Int32Type +parent : ObjectType +properties : RecordType +[AppleScript only] class : Int32Type +[AppleScript only] object reference : ObjectType</pre>
<pre>+{VB-only collection event} add(in object : ObjectType, in index : Int32Type) : VoidType[sequential] +[VB-only collection event] count() : Int32Type[sequential] +[VB-only collection event] remove(in index : Int32Type) : VoidType[sequential] +[VB/JS-only collection event] any item() : ObjectType[sequential] +[VB/JS-only collection event] every item() : ObjectArrayType[sequential] +[VB/JS-only collection event] first item() : ObjectType[sequential] +[VB/JS-only collection event] item by ID(in ID : Int32Type) : ObjectType[sequential] +[VB/JS-only collection event] item by name(in name : StringType) : ObjectType[sequential] +[VB/JS-only collection event] item by range(in from : Int32Type, in to : Int32Type) : ObjectArrayType[sequential] +[VB/JS-only collection event] item(in index : Int32Type) : ObjectType[sequential] +[VB/JS-only collection event] last item() : ObjectType[sequential] +[VB/JS-only collection event] middle item() : ObjectType[sequential] +[VB/JS-only collection event] next item(in obj : ObjectType) : ObjectType[sequential] +[VB/JS-only collection event] previous item(in obj : ObjectType) : ObjectType[sequential] +[JS-only event] get elements() : ObjectArrayType[sequential] +[JS-only event] to source() : StringType[sequential] +[JS-only event] to specifier() : StringType[sequential]</pre>

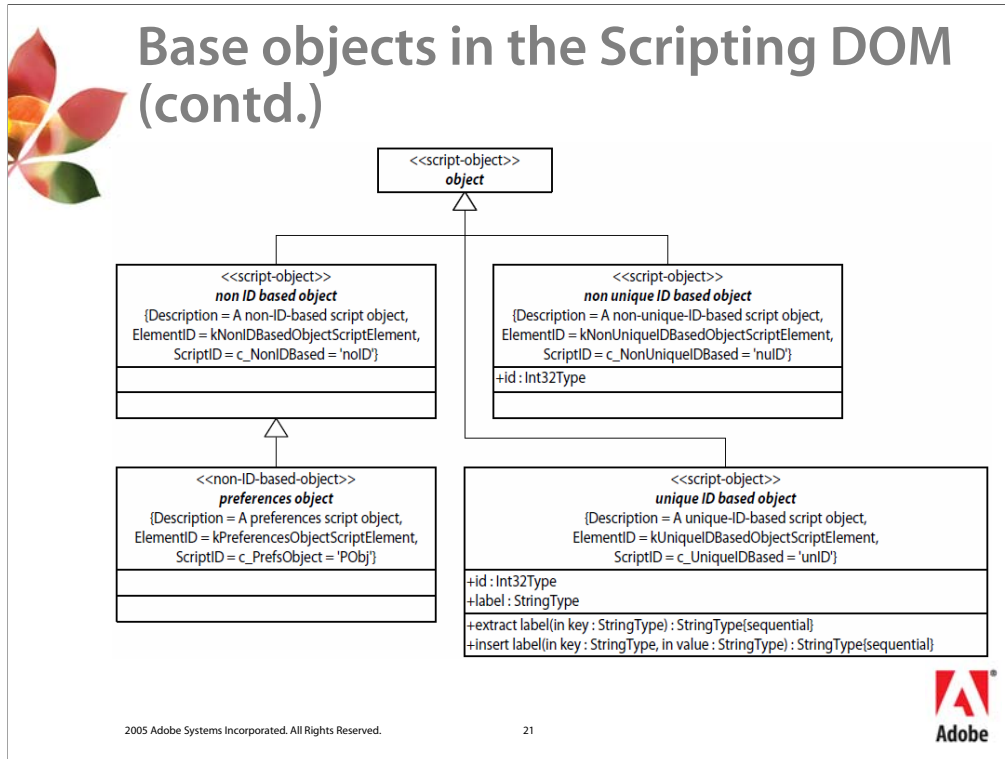
2005 Adobe Systems Incorporated. All Rights Reserved.
20


Earlier, we mentioned that one of the things you must do before making your plug-in scriptable is to determine how to represent your plug-in features in the scripting DOM. Now that we have an idea of how to specify a scripting object and element using the VersionedScriptElementInfo resource, we will now introduce you to some of the base objects in the scripting DOM that you can inherit, when specifying Objects.

At the root of the scripting DOM is the “object” object, identified by the ScriptElementID kAnyObjectScriptElement. All scripting DOM objects inherit from this object in one way or another. Just like with C++ inheritance, when a scripting DOM object inherits from another object, it also inherits the properties and events of the base object. This means that all scripting DOM objects have the properties and events shown in the UML diagram on this slide.

See also:

- Section "Script Object Inheritance" in the [Making Your Plug-in Scriptable] tech note.




Inheriting from kAnyObjectScriptElement are the following objects:

- non ID based object (ScriptElementID: kNonIDBasedObjectScriptElement): for script objects that do not have a boss object in the boss DOM.
- non unique ID based object (ScriptElementID: kNonUniqueIDBasedObjectScriptElement): for script objects that exposes a scriptable boss that has an ID of some sort but not a UID e.g. table cells.). Note that this object has the “id” property.
- unique ID based object (ScriptElementID: kUniqueIDBasedObjectScriptElement) : for script objects that expose a scriptable boss that has a UID and therefore is persistent. Note that this object has the “id” property, along with a “label” property, allowing for unique identification during run-time. (More on Script labels later in this presentation.)

Inheriting from non ID based object is the preferences object (ScriptElementID = kPreferencesObjectScriptElement), which is for preferences (generally corresponds to settings in preference interfaces in kWorkspaceBoss or kDocWorkspaceBoss).

Most scripting DOM objects inherit from one of these objects.




Scripting DOM object reference

- From [scripting-dom-javascript-idr40.html](#):
 - NonIDBasedObject (kNonIDBasedObjectScriptElement-object)**
 - ScriptID** = 'noID', **Name** = "non ID based object", **Description** = A non-ID-based script object, **BaseObject** = [kAnyObjectScriptElement](#), **Suite** = [kHiddenSuiteScriptElement](#), **Plugin** = SCRIPTING.RPLN

Event	Provider
kGetElementsEventScriptElement	kBaseObjectScriptProviderBoss
kToSourceEventScriptElement	kBaseObjectScriptProviderBoss
kToSpecifierEventScriptElement	kBaseObjectScriptProviderBoss

Property	Provider	Access
kIndexPropertyScriptElement	kBaseObjectScriptProviderBoss	read only
kParentPropertyScriptElement	kBaseObjectScriptProviderBoss	read only
kPropertiesPropertyScriptElement	kBaseObjectScriptProviderBoss	read/write

2005 Adobe Systems Incorporated. All Rights Reserved. 22



If you have ever written a script for InDesign or InCopy, you will have taken notice of the large number of objects in the scripting DOM. Furthermore, the scripting DOM varies by the context, that is, you will encounter a different scripting DOM for AppleScript, JavaScript, Visual Basic, and INX. To help you learn more about these scripting DOM objects, the SDK contains a set of HTML-based references in the {SDK}/docs/references folder, where you will find HTML files whose filenames start with "scripting-dom". If you open one of them, you will see something like what is shown on this slide.

By looking at this reference, you can find the following details about each scripting DOM object:


- The various attributes of the object, such as ScriptID, Name, Base object (inheritance), etc.
- The Events and Properties that are part of the object

This slide shows a screen shot from a portion of the JavaScript version of the scripting DOM, but there are also versions for INX, AppleScript, and Visual Basic included in the same folder. The "idr40" in the filename indicates the application, feature set and version, namely "InDesign Roman version 4.0". There is a code snippet in the SDK that allows you to generate the same reference for different feature sets, such as InDesign Japanese, InCopy Roman or InCopy Japanese, or to dynamically generate the DOM reference after adding your own scriptable plug-in. The SnippetRunner plug-in (that is used to run the code snippet that generates this reference) is scriptable (and thus shows up in the reference), however, since the UI components are part of the same plug-in, it will not load under InDesign Server.

There is also a Test menu in the debug build which allows you to generate a text dump of a scripting DOM.


See also:

- Sections "Scripting DOM Reference" and "Dumping the Scripting DOM" in the [Making Your Plug-in Scriptable] tech note.
- Code snippet SnpCreateScriptingDOMReference



Model: Scripting Architecture (contd.)

- **Scripting model refreshes dynamically at startup when a new plug-in is introduced**
 - All VersionedScriptElementInfo resources are read and Scripting DOM is generated internally
 - Macintosh: **`${home}/Library/Preferences/Adobe {APPNAME}/Version 4.0(J)/Scripting Support/Resources for AppleScript.AETE`**
 - Windows: **`%USERPROFILE%/Application Data/Adobe/{APPNAME}/Version 4.0(J)/Scripting Support/Resources for Visual Basic.tlb`**
 - There is also one in the **All Users** folder...



2005 Adobe Systems Incorporated. All Rights Reserved. 23

When the application starts up and the scripting manager finds that a new plug-in was introduced to the object model (or if the scripting support library is missing – see below), it dynamically refreshes the scripting model (comprised of script objects and script providers). Here is how it happens (in a nut-shell):

- All VersionedScriptElementInfo resources are read and the necessary Scripting DOMs (JavaScript, INX, and other platform specific DOMs below) are generated internally in memory.
- (Mac only) The AETE library is regenerated (based on the current AppleScript scripting DOM) at the path shown on the slide.
- (Win only) The COM type library is regenerated (based on the Visual Basic/COM scripting DOM) at the path shown on the slide.
- Now, if you are developing a plug-in and are changing the script object or the VersionedScriptElementInfo resource, you may need to delete the “Resources for AppleScript.AETE” or “Resources for Visual Basic.tlb” file so that the scripting model is forced to refresh.


In InDesign/InCopy CS2 for Windows, there is another .tlb file that is generated in the All Users folder. This is to allow users of InDesign who only have base user privileges in their Windows user account to run scripts. The .tlb file in this folder only gets generated upon the first time InDesign is executed on that machine, and will not be regenerated unless it has been deleted. To force this .tlb file to be regenerated, make sure you delete `C:/Documents and Settings/All Users/Application Data/Adobe/{APPNAME}/Version 4.0(J)/Scripting Support/Resources for Visual Basic.tlb`.

See also:

- Section "Reviewing Scripting Resources" in the [Making Your Plug-in Scriptable] tech note.



So far, we have only talked about the general process of making a plug-in scriptable. Let's discuss several specific development scenarios that you may encounter when you try to make your plug-in scriptable.




Scenario: Add a new property to an existing object

<<script-object>> existing object
+my property : AnyType

Generic pattern for recipe "Adding a new property to an existing script object"

2005 Adobe Systems Incorporated. All Rights Reserved. 25



Description of this scenario:

- This describes a scenario where you want to add a property to an existing scripting DOM object.

VersionedScriptElementInfo sample:


```
Property
{
    kMyPropertyScriptElement,
    p_MyProperty,
    "my property",
    "The property I am adding in",
    AnyType,
    {}
    kNoAttributeClass, // refer to attr boss if text/table attr.
}
Provider
{
    kMyScriptProviderBoss,
    {
        Object{ kSomeExistingObjectScriptElement },
        Property{ kMyPropertyScriptElement, kReadWrite },
    }
}
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from CScriptProvider. The AccessProperty method is called to handle access to this property.

See also:

- Section "Adding a new property to an existing script object" in the [Making Your Plug-in Scriptable] tech note.



Samples: Add a new property to an existing object


- **BasicPersistInterface**

<<uniqueID-based-object>> graphic object {ElementID = kGraphicObjectScriptElement}
+bpi data : StringType

<<uniqueID-based-object>> page item {ElementID = kPageItemObjectScriptElement}
+bpi data : StringType

- **See also**
 - Basic Text Adornment, CHDataMerger

2005 Adobe Systems Incorporated. All Rights Reserved. 26



Sample plug-in that implements this scenario:

- BasicPersistInterface

Description:

- Adds the “bpi data” property (StringType) to a graphic object and a page item object.

ScriptElementID(s) that are associated with this element:

- kBPIDataPropertyScriptElement

Boss class that handles this element:


- kBPIScriptProviderBoss

C++ implementation that handles this element:

- BPIScriptProvider

See also:


- BasicTextAdornment; adds the “basictextadornment shade” property (BoolType) to a text object and text style range object.
- CHDataMerger; Adds the “chdatamerger tag” property (StringArrayType) to a text object and a text style range object.
- object.



Scenario: Add a new event to an existing object

<<script-object>> existing object
+my event(in param1 : AnyType) : AnyType

Generic pattern for recipe "Adding a new event to an existing script object"



2005 Adobe Systems Incorporated. All Rights Reserved. 27

Description of this scenario:

- This describes a scenario where you want to add an event to an existing scripting DOM object.

VersionedScriptElementInfo sample:

Event // an event with two parameters: parameter 1 is required, parameter 2 is not

```

{
    kMyEventScriptElement,
    e_MyEvent,
    "my event",
    "The event I am adding in",
    Int32Type, // return type
    "Description of the return type"
    {
        p_EventParam1,
        "parameter 1",
        "The first parameter",
        StringType,
        kRequired,


        p_EventParam2,
        "parameter 2",
        "The second parameter",
        FileType,
        kOptional,
    }
}
Provider
{
    kMyScriptProviderBoss,
    {
        Object{ kSomeExistingObjectScriptElement },
        Event{ kMyEventScriptElement },
    }
}
    
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from CScriptProvider. The HandleEvent method is called to handle this event.

See also

- Section "Adding a new event to an existing script" in the [Making Your Plug-in Scriptable] tech note.




Samples: Add a new event to an existing object

- **CHDataMerger**

<pre><<non-ID-based-object>> document {ElementID = kDocumentObjectScriptElement}</pre>
<pre>+chdatamerger insert tag(in storyoffset : insertion point, in table : StringType, in field : StringType, in value : StringType) : Int32Type +chdatamerger insert tags(in storyoffset : insertion point, in file : FileType) : Int32Type +chdatamerger trace tags() : VoidType +chdatamerger merge tags(in file : FileType) : Int32Type</pre>

- **See also**
 - SnippetRunner



2005 Adobe Systems Incorporated. All Rights Reserved. 28

Sample plug-in that implements this scenario:

- CHDataMerger

Description:

- Adds the “chdatamerger insert tag”, “chdatamerger insert tags”, “chdatamerger trace tags” and “chdatamerger merge tags” events to the document object.

ScriptElementID(s) that are associated with this element:

- kCHDMIInsertTagEventScriptElement
- kCHDMIInsertTagsEventScriptElement
- kCHDMTraceTagsEventScriptElement
- kCHDMMergeTagsEventScriptElement

Boss class that handles this element:


- kCHDMScriptProviderBoss

C++ implementation that handles this element:

- CHDMScriptProvider

See also:

- SnippetRunner; adds the following events to the application object:
 - “is code snippet registered”
 - “get snip log”
 - “clear snip log”
 - “save snip log”



Scenario: Add a new object to make preferences scriptable

<<script-object>>
existing object

+my preferences : my preference object

1


1

<<preferences-object>>
my preference object

{ElementID = kMyPrefObjectScriptElement}

+my preference : AnyType

Generic pattern for recipe "Adding a new script object to make preferences scriptable"



2005 Adobe Systems Incorporated. All Rights Reserved.
29

Description of this scenario:

- This describes a scenario where you want to add an preference object to an existing scripting DOM object, which would typically be either the application or document object

VersionedScriptElementInfo sample:

```

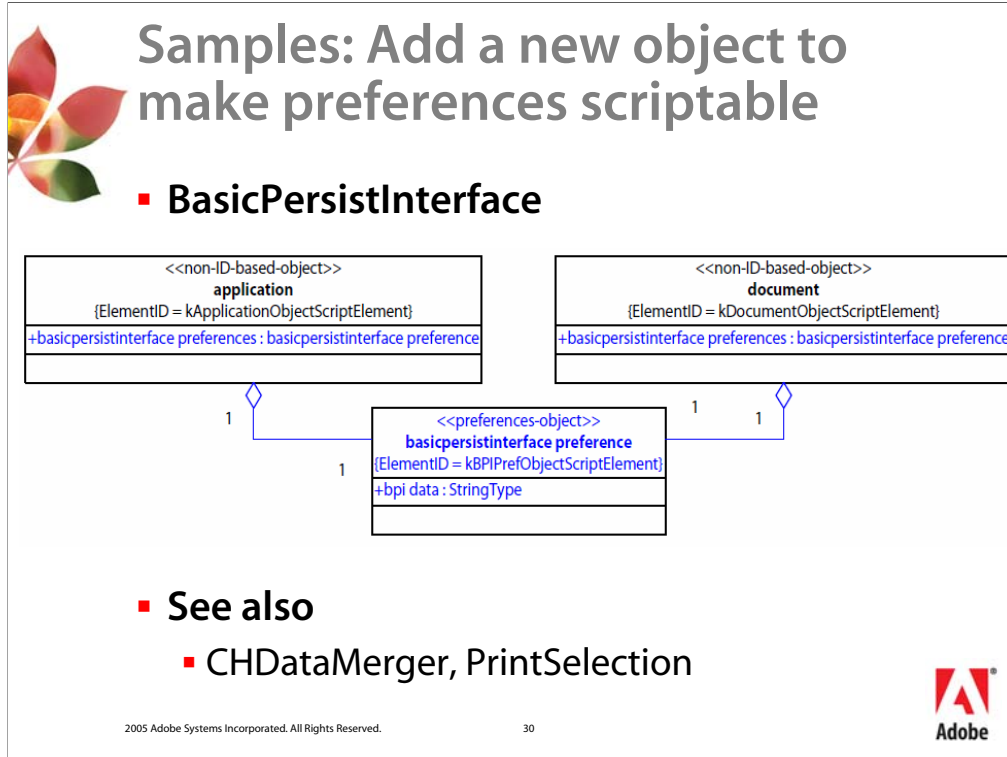
Object
{
    kMyPrefObjectScriptElement,
    c_MyPref,
    "my preference",
    "The preference object that I am adding in",
    kBPIPref_CLSID,
    NoPIuralInfo,
    kPreferencesObjectScriptElement,
    kPreferencesSuiteScriptElement,
}
Property
{
    kMyPrefObjectPropertyScriptElement,
    p_MyPref,
    "my preferences",
    "The reference to my preference object that I am adding in",
    ObjectType(kMyPrefObjectScriptElement),
    {}
    kNoAttributeClass,
}
Provider
{
    kMyScriptProviderBoss,
    {
        Parent { kApplicationObjectScriptElement }, // add to Application
        Parent { kDocumentObjectScriptElement }, // add to Document
        RepresentObject{ kMyPrefObjectScriptElement },
        ParentProperty {kMyPrefObjectPropertyScriptElement, kReadWrite},
        Property { /* see "Add a new property to existing script object" */ },
    }
}
    
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from PrefsScriptProvider. The AccessProperty method is called to handle access to the property in the preference object.

See also:

- Section "Adding a new script object to make preferences scriptable" in the [Making Your Plug-in Scriptable] tech note.



Sample plug-in that implements this scenario:

- BasicPersistInterface

Description:

- Adds the “basicpersistinterface preferences” preference object (singleton property) to the application and document objects, which refers to the “basicpersistinterface preference” object.

ScriptElementID(s) that are associated with this element:

- kBPIPrefObjectScriptElement (the preference object)
- kBPIPrefObjectPropertyScriptElement (the reference added into parent objects)

Boss class that handles this element:

- kBPIPrefsScriptProviderBoss

C++ implementation that handles this element:

- BPIPrefsScriptProvider

See also:

- CHDataMerger; Adds the “chdatamerger preferences” preference object (singleton property) to the document object, which refers to the “chdatamerger preference” object.
- PrintSelection: Adds the “printselection preferences” preference object (singleton property) to the document object, which refers to the “print selection preference” object.

Scenario: Add a new object to make UID-based bosses scriptable

The diagram shows two class-like boxes. The left box is labeled '<<script-object>> existing object' and contains a property '+my objects : my new object'. The right box is labeled '<<uniqueID-based-object>> my new object'. A blue diamond-shaped relationship symbol connects the two boxes, indicating an aggregation or inheritance relationship.

Generic pattern for recipe "Adding a new script object to make a boss that has a UID scriptable"

2005 Adobe Systems Incorporated. All Rights Reserved. 31

Description of this scenario:

- This describes a scenario where you want to add an object or a collection object whose model is represented by a persistent boss (one that has a UID) to an existing scripting DOM object.

VersionedScriptElementInfo sample:

```

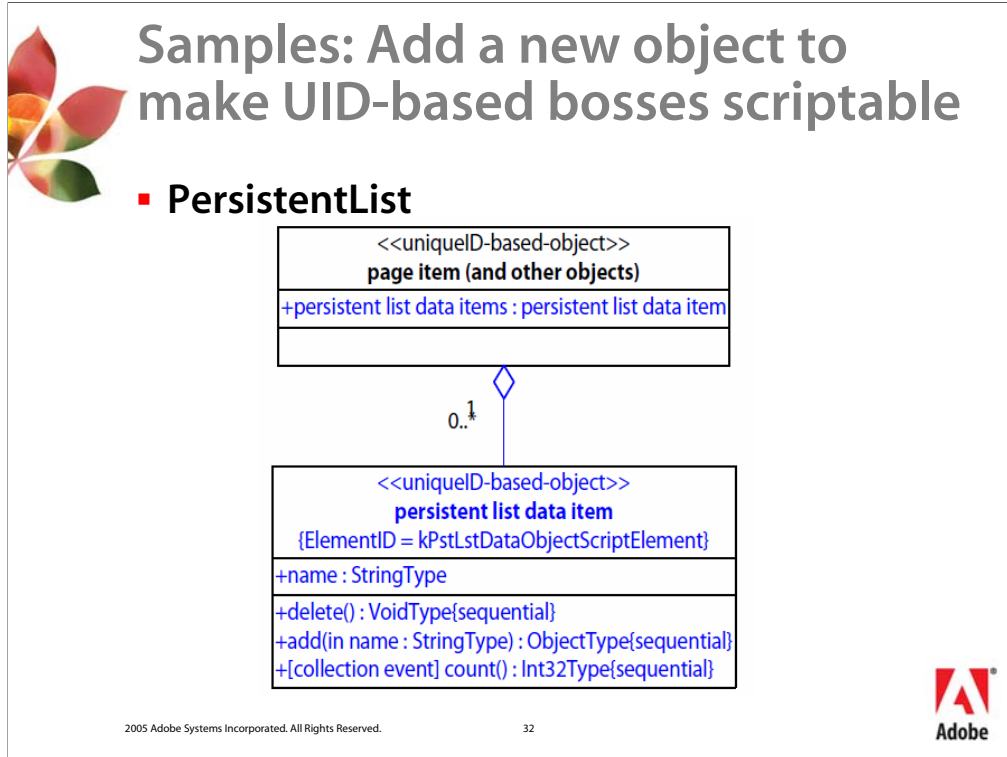
Suite // optional
{
    kMySuiteScriptElement,
    s_MySuite,
    "my suite",
    "Terms applicable to my plug-in's operations",
}
Object
{
    kMyObjectScriptElement,
    c_MyObject,
    "my object",
    "The object I am adding in",
    kMyObject_CLSID,
    c_MyObjects, // plural
    "my objects", // plural
    "A collection of objects I am adding in", // plural
    kMyObjects_CLSID, // plural
    kUniqueIDBasedObjectScriptElement, // inheritance
    kMySuiteScriptElement,
}
Provider
{
    kMyScriptProviderBoss,
    {
        Parent{ kSomeExistingObjectScriptElement },
        RepresentObject{ kMyObjectScriptElement },
        Property{ /* see "Add a new property to existing script object" */ },
    }
}
    
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from RepresentScriptProvider.
- An implementation that inherits from CScript, aggregated in your model boss class.

See also:

- Section "Adding a new script object to make a boss that has a UID scriptable" in the [Making Your Plug-in Scriptable] tech note



Sample plug-in that implements this scenario :

- PersistentList

Description:

- Adds the “persistent list data items” collection object to various objects (see PstLst.fr). Each item in the collection object refers to the “persistent list data item” object, which has a “name” property.

ScriptElementID(s) that are associated with this element:


- kPstLstDataObjectScriptElement. Note that the parent object doesn’t need to have an ScriptElementID for the added property reference, since this is added as a collection object. The fact that this object has plural attributes, combined with the specification of the Parent and RepresentObject elements in the Provider element, are enough to connect the individual objects in the collection to the parent.

Boss class that handles this element:

- kPstLstScriptProviderBoss for aggregating the IScriptProvider implementation
- kPstLstDataBoss for aggregating the IScript implementation

C++ implementation that handles this element:

- PstLstScriptProvider for the IScriptProvider implementation
- PstLstScript for the IScript implementation



Scenario: Add a new object to make a boss with no UID scriptable

<<script-object>>
existing object

+my objects : my new object


◊

<<non-uniqueID-based-object>>
my new object

Generic pattern for recipe "Adding a new script object to make a boss that has no UID scriptable"

2005 Adobe Systems Incorporated. All Rights Reserved.

33



Description of this scenario:

- This describes a scenario where you want to add an object or a collection object whose model is represented by a non-persistent boss (one that does not have a UID) to an existing scripting DOM object. This is very similar to the UID-based scenario, except that the object inherits from a different script object. See the line in bold below.

VersionedScriptElementInfo sample:

```


Suite // optional
{
  kMySuiteScriptElement,
  s_MySuite,
  "my suite",
  "Terms applicable to my plug-in's operations",
}
Object
{
  kMyObjectScriptElement,
  c_MyObject,
  "my object",
  "The object I am adding in",
  kMyObject_CLSID,
  c_MyObjects, // plural
  "my objects", // plural
  "A collection of objects I am adding in", // plural
  kMyObjects_CLSID, // plural
  kNonUniqueIDBasedObjectScriptElement, // inheritance - different from UID-based!
  kMySuiteScriptElement,
}
Provider
{
  kMyScriptProviderBoss,
  {
    Parent{ kSomeExistingObjectScriptElement },
    RepresentObject{ kMyObjectScriptElement },
    Property{ /* see "Add a new property to existing script object" */ },
  }
}
    
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from RepresentScriptProvider.
- An implementation that inherits from CScript, aggregated in your model boss class.

See also

- Section "Adding a new script object to make a boss that has no UID scriptable" in the [Making Your Plug-in Scriptable] tech note.



Scenario: Add a new object to make a C++ class scriptable

<<script-object>>
existing object

+my objects : my new object


◊

<<non-ID-based-object>>
my new object

Generic pattern for recipe "Adding a new script object to make a C++ object that has no boss scriptable"

2005 Adobe Systems Incorporated. All Rights Reserved.

34



Description of this scenario:

- This describes a scenario where you want to add an object or a collection object whose model is not yet represented by any boss to an existing scripting DOM object. This is very similar to the UID-based scenario, except that the object inherits from a different script object. See the line in bold below.

VersionedScriptElementInfo sample:

```

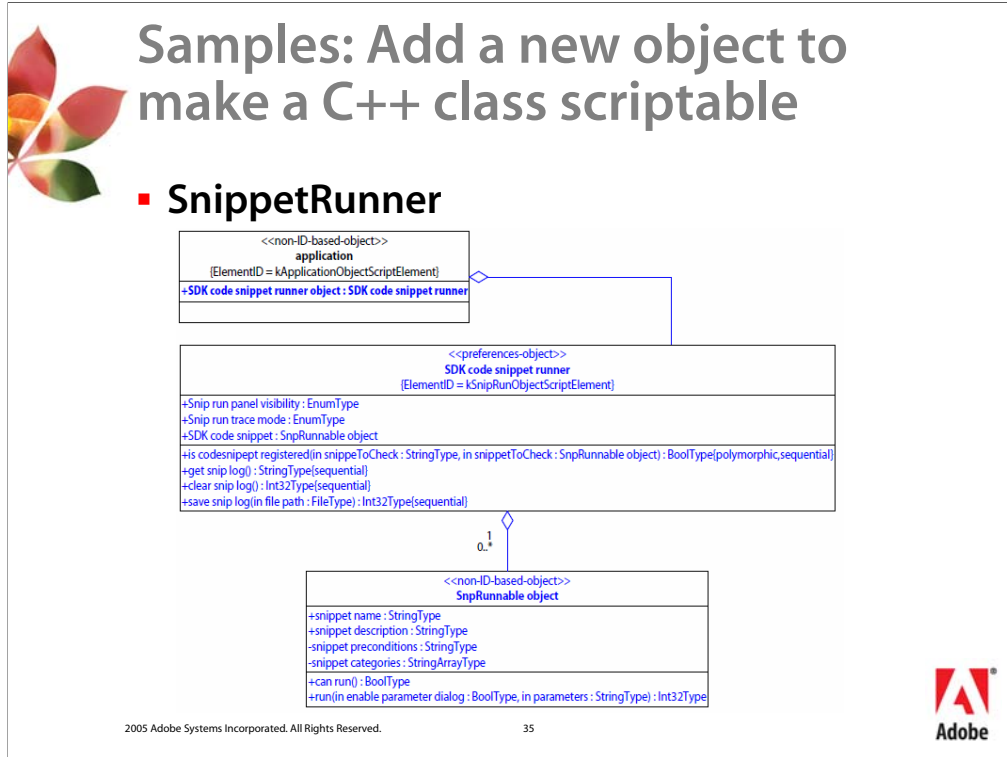
Suite // optional
{
  kMySuiteScriptElement,
  s_MySuite,
  "my suite",
  "Terms applicable to my plug-in's operations",
}
Object
{
  kMyObjectScriptElement,
  c_MyObject,
  "my object",
  "The object I am adding in",
  kMyObject_CLSID,
  c_MyObjects, // plural
  "my objects", // plural
  "A collection of objects I am adding in", // plural
  kMyObjects_CLSID, // plural
  kNonIDBasedObjectScriptElement, // inheritance - different from UID-based!
  kMySuiteScriptElement,
}
Provider
{
  kMyScriptProviderBoss,
  {
    Parent{ kSomeExistingObjectScriptElement },
    RepresentObject{ kMyObjectScriptElement },
    Property{ /* see "Add a new property to existing script object" */ },
  }
}
    
```

Required Implementations:

- A boss class that inherits from kBaseScriptProviderBoss, and aggregates an implementation that inherits from RepresentScriptProvider.
- A boss class that inherits from kBaseProxyScriptObjectBoss, and aggregates an implementation that inherits from CProxyScript.

See also:

- Section "Adding a new script object to make a C++ object that has no boss scriptable" in the [Making Your Plug-in Scriptable] tech note



Sample plug-in that implements this scenario :

- SnippetRunner

Description:

- Adds the “SDK code snippets” collection object to the “SDK code snippet runner” object, which is added in by the same plug-in. (see SnipRun.fr). Each item in the collection object refers to the “SDK code snippet” object, which represents a code snippet.

ScriptElementID(s) that are associated with this element:

- kSnpRunnableObjectScriptElement

Boss class that handles this element:

- kSnpRunnableScriptProviderBoss for aggregating the IScriptProvider implementation
- kSnpRunnableScriptObjectBoss for aggregating the IScript implementation

C++ implementation that handles this element:

- SnpRunnableScriptProvider for the IScriptProvider implementation
- SnpRunnableScript for the IScript implementation



Scenario: Adding an error string service

- **Purpose:** Associates an error string with an error code
- **Needed when a script provider needs to return an error code to the script**
 - kFailure doesn't provide the script (or user) any information about what went wrong
 - Returning specific error codes is **required** in CS2
- **How to implement**
 - Add UserErrorTable resource (in .fr file) to map error codes to string keys (with translations)
 - Implement a class that inherits CErrorStringService
 - Declare kMyErrorStringServiceBoss (a service provider)

2005 Adobe Systems Incorporated. All Rights Reserved.


36



An error string service is now required in CS2, whenever your script provider needs to report some error to the script (or user). kFailure is no longer an acceptable error to return from a script provider, as it gives absolutely no context or background information to the script or user about what went wrong in the operation.

Implementing an error string service is easy:


```
// *** in MyPlugin.fr ***
// in ClassDescriptionTable
class
{
    kMyErrorStringServiceBoss,
    kInvalidClass,
    {
        IID_ERRORSTRINGSERVICE, kMyErrorStringServiceImpl,
        IID_IK2SERVICEPROVIDER, kErrorStringProviderImpl,
    }
},
// further below.
#include "ErrorTableTypes.h"
resource UserErrorTable(kMyErrorStringResourceID)
{
    {
        kMyErrorCode, kMyErrorCodeStringKey,
        // add more pairs if needed. . .
    }
};
// don't forget to define string keys and add translations in string table . . .
// *** in MyErrorStringService.cpp ***
class MyErrorStringService : public CErrorStringService
{
public:
    MyErrorStringService(IPMUnknown* boss):
        CErrorStringService(boss, kMyPluginID,
        kMyErrorStringResourceID) {}
    virtual ~MyErrorStringService(void) {}
};
CREATE_PMIINTERFACE(MyErrorStringService, kMyErrorStringServiceImpl)
```



Testing your scriptable plug-in

- **Write a script in three languages, as various context-specific nuances exist, especially around plural objects and capitalization**
 - AppleScript
 - JavaScript
 - Visual Basic or VBScript


2005 Adobe Systems Incorporated. All Rights Reserved. 37



Capitalization and spacing rules for names are specified using the `FormatNamePolicy` class in `ScriptInfo.h` and are set in each script manager (for each scripting context). You can get the context-specific format name policy by calling `IScriptMgr::GetFormatNamePolicy`. The text in the square brackets show how the script element name "my name" (as written in the `VersionedScriptElementInfo` resource) would appear in each setting.


- AppleScript Capitalization:
 - For suites = `FormatNamePolicy::kUpperCaseFirstCharOfWords ["My Name"]`
 - For everything else = `FormatNamePolicy::kUseDefaultCapitalization ["my name"]`
- AppleScript Spacing:
 - `FormatNamePolicy::kUseDefaultSpacing ["my name"]`
- JavaScript Capitalization:
 - For events, event parameters, properties = `FormatNamePolicy::kAllLowerCaseExceptInterCaps ["myName"]`
 - For objects, enum names = `FormatNamePolicy::kUpperCaseFirstCharOfWords ["MyName"]`
 - For enum values = `FormatNamePolicy::kAllUpperCase ["MY_NAME"]`
 - For everything else: `FormatNamePolicy::kUseDefaultCapitalization ["myname"]`
- JavaScript Spacing:
 - For enum values = `FormatNamePolicy::kReplaceSpacesWithUnderScore ["MY_NAME"]`
 - For everything else: `FormatNamePolicy::kRemoveSpaces ["MyName", "myName", or "myname" depending on the element]`
- Visual Basic Capitalization:
 - For all: `FormatNamePolicy::kUpperCaseFirstCharOfWords ["MyName"]`
- Visual Basic Spacing:
 - For all: `FormatNamePolicy::kRemoveSpaces ["MyName"]`
- Visual Basic Prefix:
 - For enum names and enum values: Prefix name with "id" [`"idMyName"`]
- INX Capitalization and spacing is similar to Visual Basic, except that no prefixes are used.

(Notes continued on next page)



Writing a test script

- **AppleScript**
 - Use AppleScript Studio or your favorite AppleScript IDE
- **Visual Basic**
 - Use Visual Basic 6 (recommended)
 - Use Windows Scripting Host for VBScript
- **JavaScript**
 - The ExtendScript debugger is installed by the separate Adobe Bridge installer
 - See “Debugging JavaScript” section in [Making Your Plug-in Scriptable] tech note




2005 Adobe Systems Incorporated. All Rights Reserved. 38

Writing a test script can be done with a text editor, if you are familiar with the syntax for each scripting language, and can figure out all of the capitalization/spacing rules as mentioned on the previous slide. Scripts can be most easily written using one of the script IDEs listed on the slide.

For JavaScript, both writing and testing scripts with the application can be facilitated with the use of a debugger. The debugger of choice for InDesign CS2 is the Extend Script debugger, a full featured debugger for testing/debugging JavaScript in Adobe CS2 applications. The InDesign CS2 installer provides the option to install this, although it is somewhat hidden, as it is part of the Adobe Bridge package. The instructions are mentioned in the section “Debugging JavaScript” section in the tech note.


(Contd' from previous notes page) Some names are actually “translated” (changed) for each scripting context, due to conflicts in the associated scripting language or keyword set. This is done using the “Translation” element in the VersionedScriptElementInfo, which allows you to change a name of a script element into something else for the associated scripting context.

- AppleScript
 - “add” is changed to “make” (on many objects)
- JavaScript
 - “default” is changed to “default value” (i.e. defaultValue for enumerations)
 - “delete” is changed to “remove” (on many objects)
 - “export” is changed to “export file” (i.e. exportFile on PageItem)
 - “import” is changed to “import file” (i.e. importFile on PageItem)
 - “package” is changed to “package for print” (i.e. packageForPrint on Document)
- Visual Basic
 - “print” is changed to “print out” (i.e. “PrintOut” on Document)



Scenarios: INX support

- **Test INX round tripping using the “InDesign Snippets” format name**
 - If you want to read in INX files in InDesign CS, make sure to apply the InDesign CS 3.0.1 April 2005 updates
- **Your custom script objects must:**
 - Support the e_Create (“add”) event (no req’d params)
 - Aggregate IID_IDOMELEMENT and IID_XMLFRAGMENT
- **Custom page item objects require special care during object creation**
 - Refer to the BasicShape sample plug-in



2005 Adobe Systems Incorporated. All Rights Reserved. 39

You may be adding scriptability to your plug-in in order to have your custom persistent objects participate in the INX import/export processes.


In that case, you will probably have followed the scenario “Add a new object to make preferences scriptable” or “Add a new object to make UID-based bosses scriptable”. In the case of preferences, you have done all of the work necessary to support INX round-tripping for your custom preference data. However, if you added a new object to make your UID-based bosses scriptable, you will need to make sure to add the following:

- Your Object must support the collection event with the ID = e_Create and Name = “add” (you can reuse kCreateEventScriptElement if you wish), and you must not have any required parameters in this event, since the INX framework will first create the object without any parameters, and then based on the properties of the represented object, call AccessProperty to set each one of the properties one by one.
- The model boss (in which you added an implementation that inherits from CScript) must also include these implementations:
 - IID_IDOMELEMENT (kScriptDOMELEMENTImpl is most commonly used)
 - IID_XMLFRAGMENT (kINXNoContentFragmentImpl is most commonly used)

Custom page item objects which inherit from existing page item objects in the Scripting DOM require special care when creating the object. Refer to the BasicShape sample plug-in, specifically the method BscShpScriptProvider::HandleEvent.

Also, if you are going to be exporting an INX file (from an entire document) from InDesign CS2 and importing that into InDesign CS, you must remember to apply the “InDesign CS 3.0.1 April 2005 Updates” to your installation of InDesign CS (3.0.1). In addition, your InDesign CS plug-in must be scriptable. Most of the scenarios documented in this presentation as well as the CS2 SDK version of the [Making Your Plug-in Scriptable] do apply to CS plug-ins as well, however, there is no VersionedScriptElementInfo resource in CS; you will need to use the ScriptElementInfo resource instead.


For more details, refer to the section “Supporting INX” in the CS2 SDK version of the [Making Your Plug-in Scriptable] tech note. You can also refer to the CS SDK version of the [Making Your Plug-in Scriptable] tech note for instructions on how to make a CS version plug-in scriptable. (NOTE: You may not have to re-release your CS plug-in to add scriptability. Depending on how well you followed the boss object model and made your core components easily accessible by other plug-ins, you may just be able to publish an extra CS plug-in that provides scriptability to your original CS plug-in.)



Scenarios: InDesign Server support


- **Separate model/UI into separate plug-ins**
 - Scripting support goes into the model plug-in
- **Make your model plug-in load under the kInDesignServerProduct**
- **Make sure your VersionedScriptElementInfo includes kInDesignServerProductFS in the feature set field**
- **Your scripts should write results to a file**

2005 Adobe Systems Incorporated. All Rights Reserved. 40



Another reason you may be adding scriptability to your plug-in is to have your plug-in work under InDesign Server.


In addition to adding scriptability to your plug-in, there are some further requirements you must satisfy before your plug-in can be loaded in InDesign Server. See the [InDesign Server Plug-in Techniques] tech note for more details.



Reviewing scripting resources

- **Review the Scripting DOM**
 - Dumping the DOM to a file
 - Using VB's Object Browser or VS.NET's COM Object Browser
 - Open Dictionary with AppleScript Studio
- **Verify that your scripting resources are:**
 - Free of ScriptID/Name clashes? (at startup)
 - All included into the scripting DOM?
 - Correctly capitalization for each context?

2005 Adobe Systems Incorporated. All Rights Reserved.



Once you have satisfied the responsibilities of a scriptable plug-in, as listed in an earlier slide, worked through all of your compiler errors, and have successfully built your scriptable plug-in, you can verify if the scripting DOM elements added by your plug-in are indeed added the way you intended:


- That your scripting DOM elements are free of ScriptID/Name clashes. This is verified during the startup process in the debug build of the application. To force this verification step to happen, delete the Saved Data file from the application's preferences folder.
- That all of the scripting DOM elements you have added are added into the DOM as intended, with the intended capitalization. See previous slide for details on capitalization rules for script element names.

There are several ways to do that, as shown in this slide under the "Review the Scripting DOM" heading.

See also:

- Sections "Reviewing Scripting Resources" and "Dumping the Scripting DOM" in the [Making Your Plug-in Scriptable] tech note.






Run script from within a plug-in

```
// ClassID mgrClassID = kAppleScriptMgrBoss,
kJavaScriptMgrBoss, or
kOLEAutomationMgrBoss
InterfacePtr<IScriptManager>
scriptManager(Util<IScriptUtils>()->
QueryScriptManager(mgrClassID));
InterfacePtr<IScriptRunner>
scriptRunner(scriptManager,
UseDefaultIID());
if (scriptRunner->CanHandleFile(file))
ErrorCode err = scriptRunner->
RunFile(file, result, errorString);
```

2005 Adobe Systems Incorporated. All Rights Reserved. 43




Besides implementing script providers, there are other useful things you can do with the scripting architecture from within your plug-in.

You can run a script (as the Script panel does) from with your plug-in. This is done by:

- Querying an instance of the script manager boss, by calling `IScriptUtils::QueryScriptManager`. There are three boss class IDs to choose from, as shown on the slide. Pick one that is appropriate for your script.
- Querying for the `IScriptRunner` interface on the same boss.
- Running the script by calling `IScriptRunner::CanHandleFile` and `IScriptRunner::RunFile`.
 - `CanHandleFile` is a useful method, when you don't know the language of the script file. You can try multiple script runners (e.g. OLE and JavaScript on Windows, AppleScript and JavaScript on the Mac) and test with this method.
 - `RunFile` runs the script file, and gets two parameters back on completion:
 - Result:** This is the result from a script. In a single-file VBScript, this corresponds to the value of the "returnValue" variable, and for AppleScript and JavaScript, this corresponds to the result of the script itself.
 - ErrorString:** This is the error string from the script, if the returned `ErrorCode` is not 0. This string is associated with the error code with the aid of an error string service.

Q: Why is this use case useful?


- Some features cannot be automated using the C++ API, but can be with scripting (e.g. Package feature).
- You can dynamically generate a script (since it's just a text file) and run it from your plug-in! The sky is the limit.



Iterate document structure

- **...using the Scripting DOM!**
 - Iterate using IDOMEElement
 - IDOMEElement identifies a “DOM element”
 - Start at IDOMEElement in kDocBoss
 - Traverse the DOM tree using IDOMEElement::GetChildElements
 - Alternative to navigating document structure with IHierarchy
 - This is how these InDesign CS2 features generate their content
 - Export provider for InDesign interchange format
 - Package for Go Live CS2

2005 Adobe Systems Incorporated. All Rights Reserved. 44




You can iterate the document structure using the Scripting DOM, by utilizing the IDOMEElement interface. You would start at IDOMEElement on kDocBoss, and traverse the Scripting DOM tree.

•IDOMEElement identifies a DOM element, or an item in the Scripting DOM. This usually is aggregated together with IScript. (See kBaseProxyScriptObjectBoss, for example.)


Why is this use case useful?

•Iterating document content using IDOMEElement allows you uniform access to all objects that are relevant to a document, not just spread layers, page items and pages (like you navigate with IHierarchy). Objects like Text on a Path and Inline Graphics require you to go thru a different hierarchy than the document structure hierarchy.



Using script labels

- **Get script label from an UniqueID-based item**
 - Query for IScript from a DOM element
 - IScript is aggregated on many bosses!
 - IScript::GetTag
- **Set script label on an UniqueID-based item**
 - Call IScriptUtils::SetScriptingTag or IScriptUtils::SetScriptingTags
 - Processes kSetScriptingTagCmdBoss internally
 - Notification on ISubject of kDocBoss (protocol = IID_ISCRIPT)



2005 Adobe Systems Incorporated. All Rights Reserved. 45

You can get a script label (also known as a “script tag”) by querying for IScript from a DOM element (there are many boss classes that aggregate IScript – see HTML-based reference documentation – some you will recognize...), and then calling IScript::GetTag. You can either get the default tag value, or you can get the value of a key-value pair by passing in the key.

You can also set a script label by calling IScriptUtils::SetScriptingTag or IScriptUtils::SetScriptingTags. (kSetScriptingTagCmdBoss is still available. The data interface is IScriptTagCmdData).

To populate a ScriptList (K2Vector<IScript>), query the IScript interface on the items you want to modify labels, and call `push_back` to add them to the vector.

To get the RequestContext:


```
InterfacePtr<IScriptManager>scriptMgr(Utils<IScriptUtils>()-
>QueryScriptManager(kScriptTagMgrBoss));
```

```
RequestContext context = scriptMgr->GetRequestContext();
```

The ScriptLabelKeyValueList is a K2Vector<ScriptLabelKeyValuePair>. See IScriptLabel.h.

Why is this use case useful?


- You can put some simple string data on any scripting DOM object (e.g. page item) and later identify it uniquely. Even if the UID of an object changes (e.g. when you copy/paste or move it to a different document), the script label goes with the object.



Other miscellaneous use cases

- **Get path to scripts folder**
 - IScriptsPanelUtils::GetScriptsFolder
- **Convert 4-letter ScriptID to PMString**
 - IScriptUtils::GetScriptID (ScriptID scriptID)
 - See Session 10: Graphics (Use case: Getting graphic file type)
- **Get scripting support folder**
 - Where the COM Type Library or AETE Library is stored
 - IScriptUtils::GetScriptingSupportFolder
- **See methods on IScriptUtils and I***ScriptUtils**

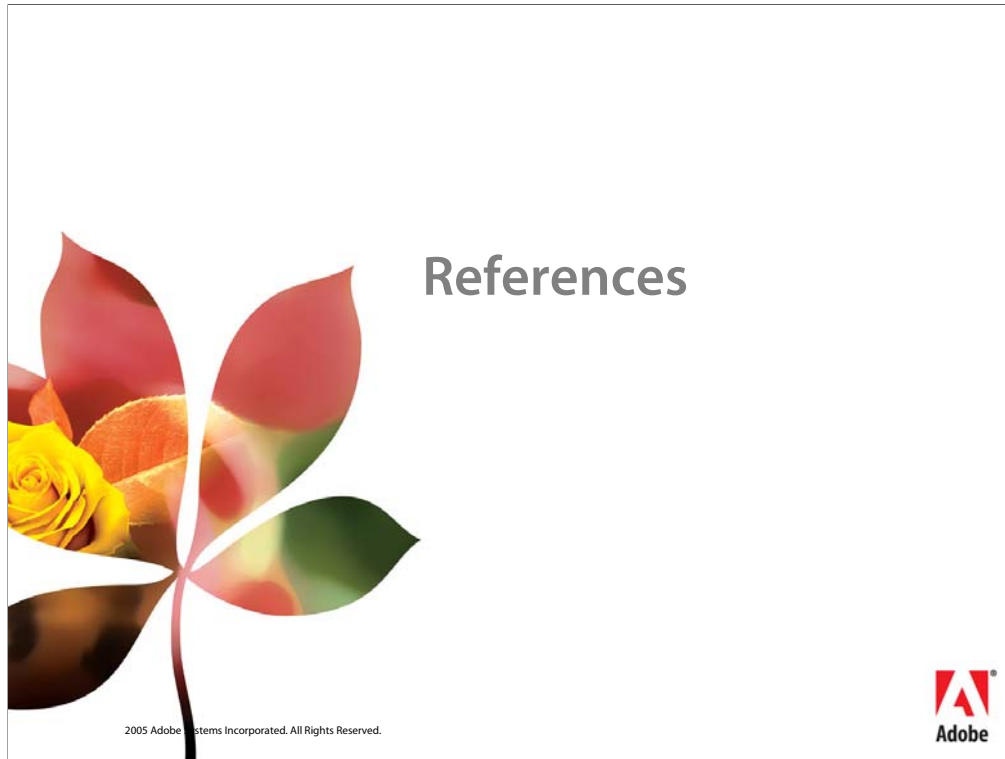
2005 Adobe Systems Incorporated. All Rights Reserved. 46




•You can get the path to the Scripts folder (the folder that the “Scripts” panel looks at for scripts) by calling IScriptsPanelUtils::GetScriptsFolder.

•You can convert a 4-letter ScriptID value to a PMString by calling IScriptUtils::GetScriptID. This is useful because the byte order of the ScriptID literals can be confusing depending on which platform (Win or Mac) your plug-in is running on.

•You can also get the path of the Scripting support folder by calling IScriptUtils::GetScriptingSupport.






SDK Documentation


- **CS2 SDK Tech Notes**
 - Making Your Plug-in Scriptable (scriptableplugin.pdf)
 - Shared Application Resources (shared-application-resources.pdf)
 - InCopy Interchange Format (incopyinterchangeformat.pdf)
 - Working with the INX file format (ww-inx-file-format.pdf)
 - InDesign Server Plug-in Techniques (indesign-server-plugin-techniques.pdf)
- **CS2 Programming Guide**
 - “Snippet Fundamentals” chapter

2005 Adobe Systems Incorporated. All Rights Reserved. 48



The following documents are available in the CS2 SDK for further exploration.


- The [Making Your Plug-in scriptable] tech note is the definitive guide in the SDK that discusses how to add scriptability to your plug-in.
- The [Shared Application Resources], [InCopy Interchange Format] and [Working with the INX file format] tech notes and the Snippet Fundamentals chapter in the [Programming Guide] all discuss various aspects of INX support in InDesign and InCopy, in detail.
- The [InDesign Server Plug-in Techniques] tech note discusses the steps necessary to prepare your plug-ins for use with InDesign Server. Adding scriptability may be one of the necessary steps for your plug-in.




SDK Documentation (contd.)

- **CS2 Scripting DOM References for C++ Programmers**
 - AppleScript (scripting-dom-applescript-idr40.html)
 - JavaScript (scripting-dom-javascript-idr40.html)
 - INX (scripting-dom-inx-idr40.html)
 - COM (scripting-dom-visualbasic-idr40.html)

2005 Adobe Systems Incorporated. All Rights Reserved. 49




These reference documents in the CS2 SDK show the Scripting DOM for the various scripting contexts and have been automatically generated from a code snippet with the help of some XSL transformation. These are great references to use when determining where to add your scripting elements.



Sample code

- **Scriptable plug-ins**
 - BasicPersistInterface
 - BasicShape
 - BasicTextAdornment
 - CHDataMerger
 - PrintSelection
 - PersistentList
 - SnippetRunner
- **Code snippets**
 - SnpCreateScriptingDOM Reference
 - Others mentioned in this presentation, to be provided in an update to the SDK

2005 Adobe Systems Incorporated. All Rights Reserved. 50



The following sample plug-ins and code snippets are available in the CS2 SDK for further exploration.



Summary

- **After this session, you should be ready to add scriptability to your plug-ins for:**
 - Scripting support
 - INX support
 - InDesign server support

