

Chapter 1: Developing and loading sub-applications

About loading sub-applications

Flex lets you load and unload sub-applications in a main application. Reasons to use sub-applications as part of your overall application architecture include the following:

- Reduce the size of the main application
- Encapsulate related functionality into a sub-application
- Create reusable sub-applications that can be loaded into different applications
- Integrate third-party applications into your main application

The way in which the sub-application is loaded defines the level of interoperability between the main application and its sub-applications. Consider the following factors when loading sub-applications in your main application:

Trusted applications What level of trust do the applications have? Trusted applications have a greater amount of interoperability with the main application. Untrusted applications, while limited, can still interoperate with the main application in some ways. In general, though, if you do not have complete control over the development and deployment of a loaded application, consider that application to be untrusted.

Versioning Are the applications compiled with the same version of the Flex framework? The default method of loading sub-applications assumes that all applications are compiled by the same version of the framework. However, Flex can load applications that were compiled with different versions of the framework. These applications are known as *multi-versioned* applications. Multi-versioned applications have some restrictions on their level of interoperability with the main application that loads them. They are, however, more flexible to use in a large application.

The level of trust and use of versioning are determined by the application domain and the security domain into which the sub-applications are loaded.

There are three main types of loaded sub-applications in Flex:

Sandboxed applications are loaded into their own security domains, and can be multi-versioned. Using sandboxed applications is the recommended practice for loading third-party applications. In addition, if your sub-applications use RPC or DataServices-related functionality, you should load them as sandboxed.

Multi-versioned applications can be compiled with different versions of the Flex framework than the main application that loads them. Their interoperability with the main application and other sub-applications is more limited than single-versioned applications.

Single-versioned applications are guaranteed to have been compiled with the same version of the compiler as the main application. They have the greatest level of interoperability with the main application, but they also require that you have complete control over the source of the sub-applications.

Using sub-applications is in some ways like using modules. For a comparison of the two approaches, see “Comparing loaded applications to modules” on page 13.

For more information, see “Developing sandboxed applications” on page 31 and “Developing multi-versioned applications” on page 39.

About application domains

An application domain is a container for class definitions. Applications have a single, top-level application domain called the system domain. Application domains are then defined as child nodes of the system domain. When you load a sub-application into another, main application, you can load it into one of three application domains: sibling, child, and current. When you load a sub-application into a *sibling* application domain, the sub-application’s application domain has the same parent as the main application’s application domain. In addition, it is a peer of all other sibling applications. When you load a sub-application into a *child* application domain, the sub-application’s application domain is a child of the main application’s application domain. When you load a sub-application into a *current* application domain, the sub-application is loaded into the same application domain as the main application. Each of these locations defines where the sub-application can get its class definitions from.

The default behavior of the SWFLoader and Loader controls is to load a sub-application into a *child* application domain. If the sub-application and the main application are compiled with different versions of the Flex framework, runtime errors can result. These errors occur because the applications are sometimes compiled against different definitions of the same classes.

You can specify that the main application loads a multi-versioned sub-application. It does this by loading the sub-application into a *sibling* application domain. This means that the sub-application defines its own class definitions and does not get them from its parent. It is possible for two applications to work together, even if they are compiled with different versions of the Flex framework.

You specify the application domain of a sub-application by setting the value of the `loadForCompatibility` property on the SWFLoader. If you set the value of this property to `true`, then the sub-application is loaded into a sibling application domain. If you set the value of this property to `false`, then the sub-application is loaded into a child application domain. The default value of this property is `false`, so by default, sub-applications are not multi-versioned.

You can also specify the application domain on the LoaderContext object. You do this if you specify the value of the `loaderContext` property when using the SWFLoader control. For more information, see “Specifying a LoaderContext” on page 21.

The system domain

Classes defined by Flash Player are in the system domain. The system domain parents all other application domains. The main application’s application domain is a child of the system domain. If you load sub-applications into sibling application domains, then they are also children of the system domain. Classes defined in the system domain are never redefined in sub-applications or main applications. Those applications all share the common definitions of Flash Player. These definitions include classes such as DisplayObject, Event, and Sprite. The definitions of these shared classes are contained in the `playerglobal.swc` file.

Sibling application domains

The application domain that a sub-application is in determines where the sub-application gets its class definitions from. If the main application loads a sub-application into a sibling application domain, the sub-application defines its own non-player class definitions. This is the configuration for multi-versioned applications. Applications that are loaded into a sibling application domain can load other applications into a sibling application domain.

sub-applications that are in sibling application domains of the main application can communicate with the main application. Sub-applications can call methods and access properties on the applications, as long as they meet these criteria:

- Only strong types that are defined in Flash Player are used.
- The sub-applications are not in different security domains.

The ability for a sub-application in a sibling application domain to communicate with the main application is not as seamless as when the sub-application is in a child application domain. For example, if the sub-application launches a pop-up control, it passes an event to the main application that actually launches the control. This event passing limits their communication somewhat, but does provide some measure of interoperability. For more information about developing this type of application, see “Developing multi-versioned applications” on page 39.

Applications that are in separate security domains are automatically in separate, sibling application domains. As a result, if you load an untrusted sub-application into a main application, that sub-application has its own class definitions.

For more information, see “Developing multi-versioned applications” on page 39.

Child application domains

If a main application loads a sub-application into a child application domain of its application domain, the sub-application gets its class definitions from the main application. This behavior is the default for application loading. It can result in runtime errors if the applications are compiled with different versions of the Flex framework. As the SWF file sets up its classes, classes that are already defined are not added to the application domain. First in wins, which means that the first class defined becomes the only definition of that class. Subsequent definitions loaded into that application domain are ignored.

Sub-applications that are in child application domains of the main application can communicate with the main application. They have the highest level of possible interoperability with the main application. This situation is typical of a large application that is not multi-versioned, and it is the default behavior for the SWFLoader.

For more information, see “Creating and loading sub-applications” on page 17.

The current application domain

If you load a sub-application into the current application domain (rather than a separate, sibling application domain or a child application domain), the sub-application’s class definitions are often ignored. This behavior is because the first definition in a domain is used. Subsequent definitions loaded into that domain are ignored. If new class definitions are added, the main application can use them.

Using the current application domain is typical of RSLs and other specially compiled resources, and is not typically used when loading sub-applications.

About security domains

Security domains define the level of trust between applications. The greater the trust between applications, the greater the amount of possible interoperability between those applications. In general, if a sub-application is loaded into the same security domain as the main application, then the applications have the highest level of interoperability.

If a sub-application is loaded into a different security domain (as is the case with many remote or multi-versioned applications), then the sub-application is allowed a limited amount of interaction with the main application. Sub-applications in separate security domains are also restricted in their ability to communicate with one another. They are known as sandboxed applications.

You determine whether a sub-application is loaded into the same security domain as the main application when you load it. You can set the value of the `trustContent` property to `true` to load a remote sub-application into the same security domain as the main application. This behavior only applies if that application is loaded from a different web domain or subdomain than the main application. If the sub-application is loaded from the same web domain as the main application, then it is by default loaded into the same security domain. Setting the value of the `loadForCompatibility` property does not affect the `trustContent` property.

In some cases, you want to load a sub-application that is on the same domain as the main application into a separate security domain. This might be because the sub-application is from a third party or you want your applications to have the same level of interoperability as a sandboxed application. One way to do this is to set up a different sub domain name on the same server, and load the sub-application from that sub domain. For more information, see “Loading same-domain and cross-domain applications” on page 15.

When using AIR, you cannot set the value of the `trustContent` property to `true`.

If you do not set the value of the `trustContent` property to `true`, then a sub-application on a remote server is loaded into a separate security domain by default.

You can also specify the security domain on the `LoaderContext` object. You do this if you specify the value of the `loaderContext` property when using the `SWFLoader` control. You can only do this if you want to load the sub-application into the same security domain. For more information, see “Specifying a `LoaderContext`” on page 21.

Sandboxed applications have the greatest number of limitations on application interoperability. These restrictions include the following:

Stage Access to the stage from the sub-application is limited to some stage properties and methods.

Mouse You cannot receive mouse events from objects in other security domains.

Pixels Applications cannot access the pixels drawn in applications that are in other security domains.

Properties While applications can get references to objects in other security domains, avoid doing this for security reasons. Some properties are restricted, such as the `Stage` or any parent of a `DisplayObject` that another application instantiates. In addition to these restrictions, applications that are in separate security domains are also in separate application domains by definition. As a result, they are subject to all restrictions of that architecture. However, they can also benefit from this situation because they can then be multi-versioned.

For more information about working with sandboxed applications, see “Developing sandboxed applications” on page 31

Common types of applications that load sub-applications

Common types of applications that load sub-applications include:

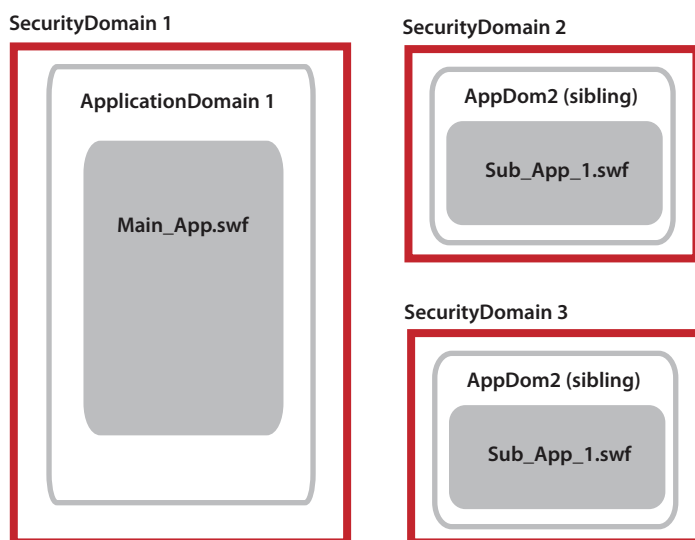
- Large applications that are or are not multi-versioned
- Mashups
- Portals
- Dashboards

About sandboxed applications

A sandboxed application is a common type of application that loads sub-applications. In general, sandbox applications load applications that are compiled and hosted by third parties. The main application does not necessarily trust the third party that developed those sub-applications. In addition, the developer of the main application does not know the version of the Flex framework that was used to compile the sub-applications. As a result, sandboxed applications have the least amount of interoperability between the main application and the sub-applications, but they are typically multi-versioned. A common type of sandboxed application is a portal.

Sandboxed applications require the least amount of additional coding when using RPC classes and DataServices-related functionality. Trusted multi-versioned applications often require a bootstrap loader so that RPC classes work across applications. Sandbox applications do not have this requirement. As a result, with many multi-versioned applications, using sandboxed applications is actually the recommended approach.

The following image shows the boundaries of a sandboxed application. The sub-applications are loaded into separate, sibling application domains. This means that each application contains its own class definitions so they can interoperate with applications that were compiled with different versions of the framework. The sub-applications are also in separate security domains. This separation means that they do not trust each other, and therefore have limited interoperability.



For more information, see “Developing sandboxed applications” on page 31.

About large, multi-versioned applications

A large, multi-versioned application is typically made up of many different components that are developed by different groups within an organization. For example, a form manager.

Multi-versioned applications load the sub-applications into a sibling application domain that is alongside the main application’s application domain. They also load sub-applications into the same security domain as the main application.

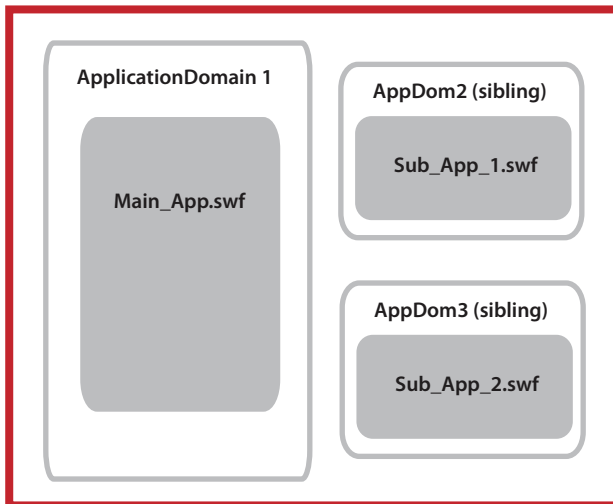
Multi-versioned applications have many of the benefits of sandboxed applications, but can require additional coding to work with RPC classes and DataServices-related functionality. In general, if your sub-applications use these classes, use the sandboxed application approach for loading sub-applications, even if they are trusted. For more information, see “Using RPC and DataServices classes with multi-versioned applications” on page 40.

In deployment, the main application and the sub-applications are typically all in the same web domain, and so they have a trusted relationship. Even if they are deployed on different web domains, the applications are typically loaded into the same security domain to maintain the trusted relationship. This process of loading cross-domain applications into the same security domain is known as *import loading*. Use this process only under rare circumstances where you know that you can trust the source of the loaded sub-applications.

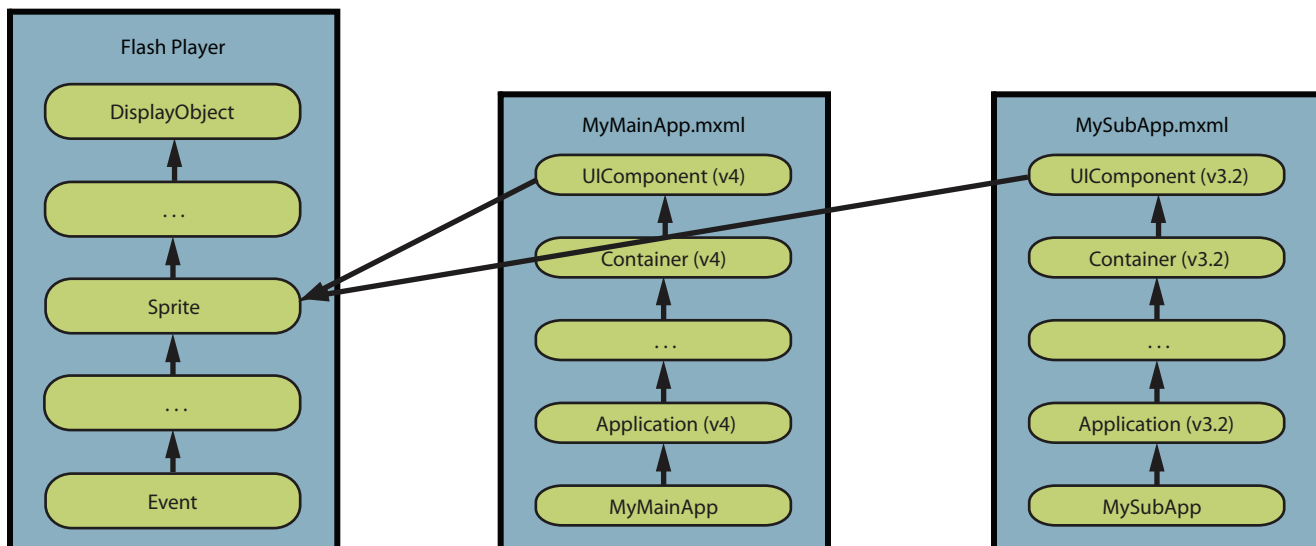
Sometimes third parties compile the sub-applications that the main application loads. In this case, developers sometimes don't know the version of the Flex framework that was used to compile the sub-applications. They might not have access to the source code of the sub-applications. As a result, they would not be able to recompile them with the same version of the framework as the main application.

The following image shows the boundaries of a large, multi-versioned application. The sub-applications are loaded into separate, sibling application domains. This means that each application contains its own class definitions so they can be compiled with different versions of the framework. All the applications are loaded into the same security domain and therefore trust each other.

SecurityDomain 1



Sub-applications in a sibling application domain store their own class definitions, apart from the class definitions in the main application. The following image shows the class definitions of a large, multi-versioned application. You can see that the sub-application uses its own definitions, which were compiled with Flex 3.2). The main application uses its own definitions, which were compiled with Flex 4.



For more information, see “Developing multi-versioned applications” on page 39.

About large, single-versioned applications

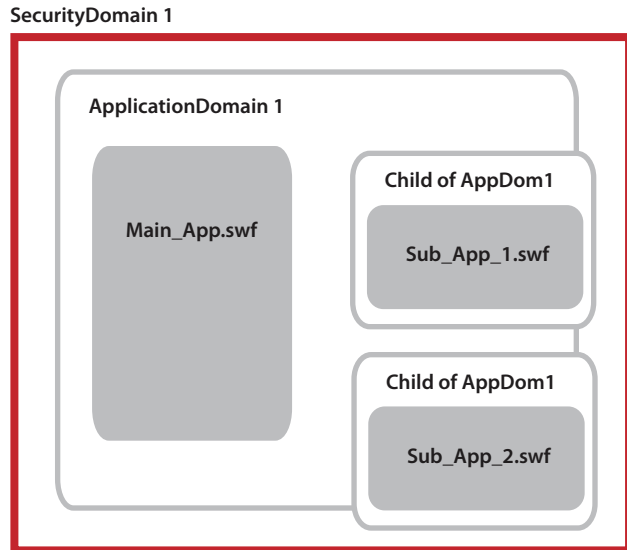
A large application without versioning support is an application that loads sub-applications that must have been compiled with the same version of the Flex framework. A single group within an organization typically creates these applications. It is possible for that group to enforce Flex framework versions and other standards during the development process.

Large, single-versioned applications are the default type of application loaded by the SWFLoader control. If you accept the SWFLoader’s default settings when you load sub-applications into your main application, the sub-applications are not multi-versioned. They are loaded into child application domains of the main application.

It can be difficult to maintain large, single-versioned applications. This is because all sub-applications must be recompiled whenever any of the applications are recompiled with a new version of the framework. In addition, it is more difficult to add third-party sub-applications because they might have been compiled with different versions of the framework. In many cases, you do not have access to the source code to recompile them.

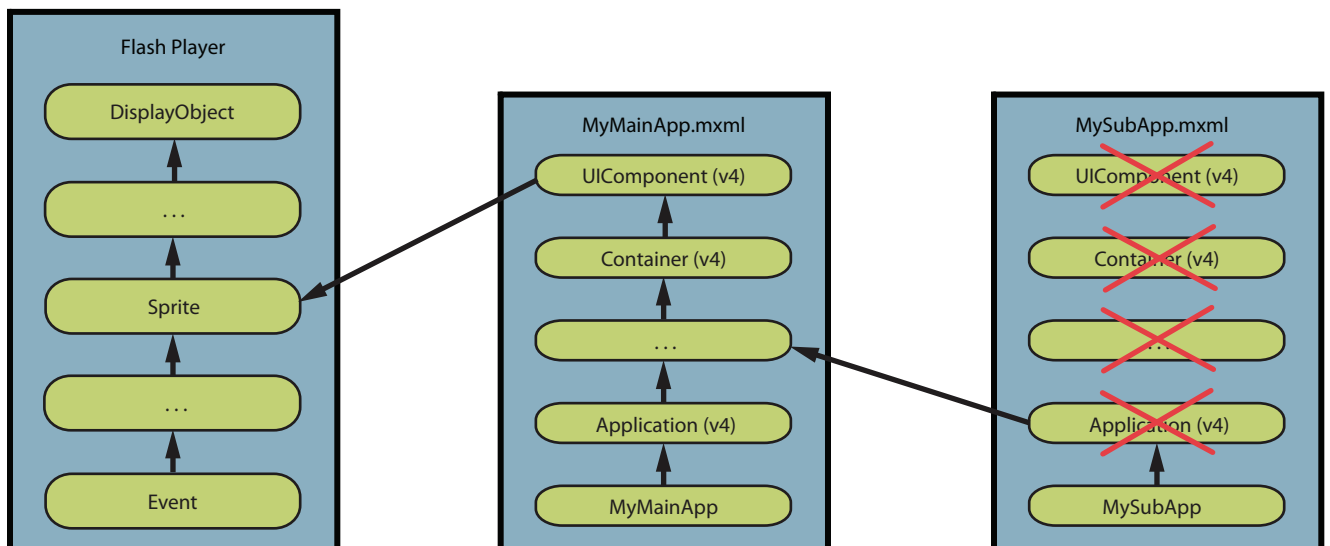
Large, single-versioned applications and all their sub-applications are usually deployed in the same web domain. This is because the same group within an organization typically develops and maintains them. As a result, they have a trusted relationship. Even if the applications are deployed on different web domains, the applications would be import loaded into the same security domain. This results in the applications having a trusted relationship.

The following image shows the boundaries of a large application that does not support multi-versioning. Each sub-application is loaded into a child application domain of the main application's application domain. As a result, all the applications use the same class definitions. If one of these applications is compiled with a different version of the framework, then runtime errors are likely to occur when a call is made to an API that is different. All the applications are inside the same security domain and therefore trust each other.



sub-applications in a child application domain inherit their class definitions from the application in the parent application domain. If a sub-application defines one of the classes that is already defined in the main application, the child's definition is ignored. If multiple sub-applications define the same class that isn't defined in the main application, each sub-application uses its own definition.

The following image shows the class definitions of a large, single-versioned application. In this image, you can see that the sub-application gets its definitions from the main application, which was compiled with Flex 4. The class definitions in the sub-application, which was also compiled with Flex 4, are ignored.



For more information, see "Creating and loading sub-applications" on page 17.

About the Flex manager classes

The Flex manager classes handle various tasks of the Flex application. For example, the `DragManager` handles all the drag-and-drop functionality; this manager class is responsible for marshaling data, creating the `DragProxy`, and triggering drag-related events.

When sub-applications are loaded into the same application domain as the main application, the application domain contains only a single instance of each manager. When applications are in different application domains, though, there can be more than one instance of a manager in the system domain.

Depending on the type of task, the manager classes are sometimes allowed to communicate through event passing when a main application loads a sub-application. For example, the `FocusManager` in a sub-application receives control from the `FocusManager` in the main application when the focus shifts to the sub-application. When the focus shifts away from the sub-application, the sub-application's `FocusManager` passes control back to the main application's `FocusManager`.

In other cases, there can only be one active instance of a manager in the system domain. Flex ensures that the sub-application's manager is disabled. For example, the `BrowserManager` cannot have multiple instances. Only the main application can access it, and only the main application can communicate with it, unless the sub-application is in a child application domain of the main application. If a sub-application in a separate application domain wants to communicate with the `BrowserManager`, it must do so through the main application.

In this case you would have to create custom logic that handles interaction with manager classes. For example, if you want a sub-application to use the main application's deep linking, you can create a custom class. This custom class passes messages from the sub-application to the main application, where the `BrowserManager` can be communicated with.

What typically happens is that the top-level manager handles the user interaction. This manager receives messages from the sub-application's manager that instructs it on what to do.

The following manager classes are linked in by all applications:

- `SystemManager`
- `LayoutManager`
- `StyleManager`
- `EffectsManager`
- `ResourceManager` (which links in `ModuleManager`)
- `FocusManager`
- `CursorManager`
- `ToolTipManager`

The following manager classes are linked in only when used in an application:

- `DragManager`
- `BrowserManager`
- `HistoryManager`
- `PopUpManager` (note that if a sub-application uses pop-up controls, the main application must include a definition of this manager)

If a main application loads sub-applications that are compiled with the same version of the framework, only one definition of each manager is stored in the SWF file. However, if you have a sandboxed or multi-versioned sub-application, both the main application and the sub-application store their own definitions. In many cases, the manager classes in the sub-application pass messages to the main application's instance of the manager. The main application's manager can then handle the task.

With sandboxed or multi-versioned applications, both the main application and the sub-application have their own versions of the manager classes. In single-versioned applications, only one version of most manager classes is necessary. You can externalize the definition of a manager in many cases for applications that are not multi-versioned.

About the `SystemManager` class

All applications have an instance of the `SystemManager` class. If an application is loaded into another application, a `SystemManager` for the sub-application is still created. The `SystemManager` still manages the sub-application's application window, but it might not manage pop-up controls, tool tips, and cursors, depending on the way in which the sub-application was loaded. The content of the `SWFLoader` that loaded the sub-application contains a reference to the sub-application.

The `SystemManager` of the main application is important because it gives you access to many aspects of the entire application. For example, the top-level `SystemManager` does the following:

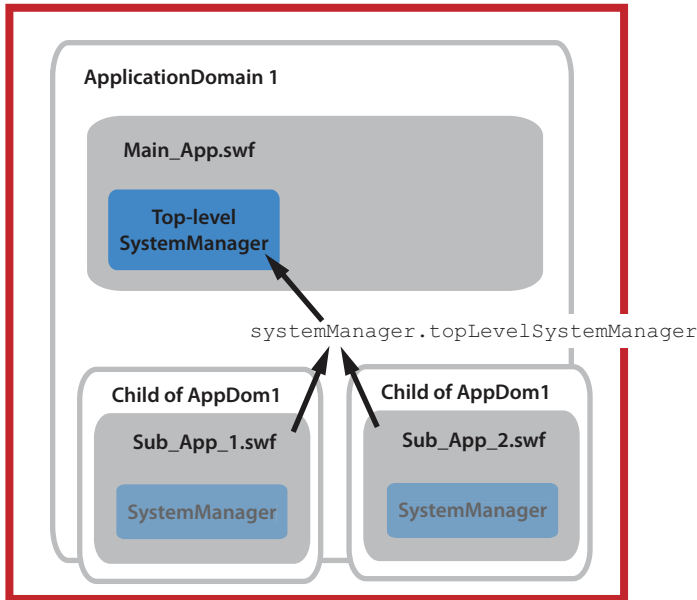
- Parents all pop-up controls, `ToolTip` objects, and cursors in the main application and all trusted sub-applications
- Handles focus for all applications that are trusted

You can use a reference to the top-level `SystemManager` to register to listen for events in a sub-application. For example, your sub-application can listen for mouse events that are outside its application domain by adding listeners to the top-level `SystemManager`.

The way you access the top-level `SystemManager` from a sub-application depends on the type of sub-application that you are using.

To get access to the top-level SystemManager in an architecture where sub-applications are loaded into child application domains, you use the `topLevelSystemManager` property of the SystemManager. The following image shows that the sub-applications use the `systemManager.topLevelSystemManager` to access the SystemManager in the main application.

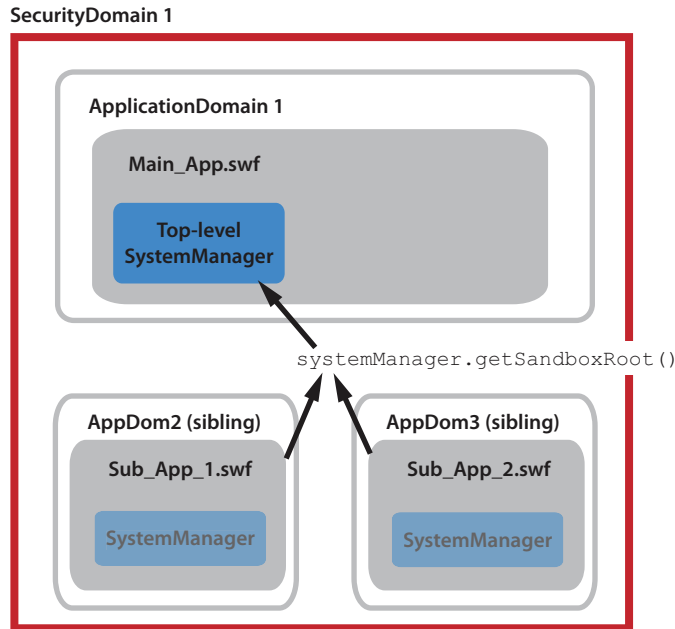
SecurityDomain 1



For a code example that uses the `topLevelSystemManager` property to access the main application's SystemManager, see "Listening for mouse events with loaded applications" on page 27.

For single-versioned applications, the `topLevelSystemManager` property always refers to the top-level SystemManager in an application domain. If your applications are in separate application domains (as sandboxed or multi-versioned applications are), use the `getSandboxRoot()` method to get the top-level SystemManager for that security domain.

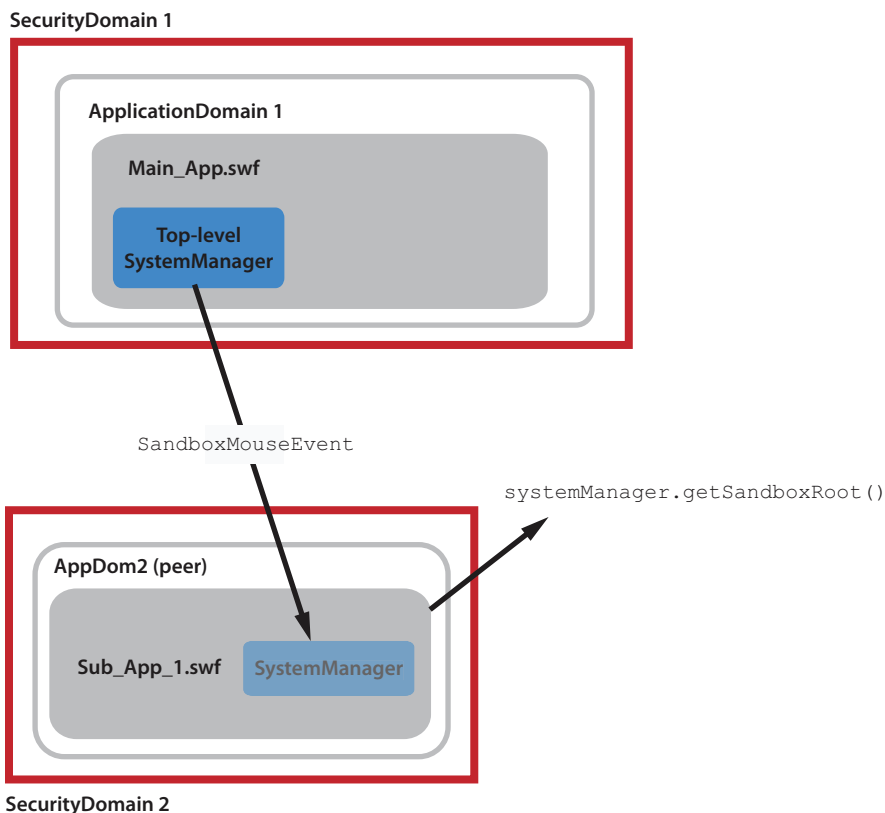
The following image shows that the sub-applications use the `systemManager.getSandboxRoot()` to get a reference to the main application's `SystemManager`, which is in a separate application domain.



For a code example that uses the `getSandboxRoot()` method to access the main application's `SystemManager`, see “Listening for mouse events in multi-versioned applications” on page 42.

When a sub-application is in a different security domain as the main application (or *not trusted*, such as in a sandboxed application), you cannot get a reference to the main application's `SystemManager` from the sub-application. In this case, you can listen for a `SandboxMouseEvent` and `InterDragManagerEvent` for inter-application communication. Any application can trigger these events. The sub-application's `SystemManager` gets notification of them, at which point you can handle them in the sub-application.

The following image shows an architecture where two applications, in separate security domains, communicate through event passing. The sub-application calls the `getSandboxRoot ()` method to get a reference to its own `SystemManager`. The `SystemManager` can then listen for events that the main application's `SystemManager` dispatched.



The properties of a `SandboxMouseEvent` object are not the same as the properties in a typical event object. For example, the `target` and `currentTarget` properties are the `SandboxBridge` that dispatched the event, or the `SystemManager` that redispached it. These properties are not the specific object that dispatched the event. Untrusted child applications should not be able to get references to objects that are typically specified by these properties. Similarly, the `stageX`, `stageY`, `localX`, and `localY` properties are not available.

For an example that uses the `SandboxMouseEvent` object to handle mouse events outside a sub-application's security domain, see "Listening for mouse events with sandboxed applications" on page 35.

Comparing loaded applications to modules

When you design a large application, give some consideration to its architecture. If the application has one main application that loads and unloads subordinate applications, then weigh the benefits of using either modules or sub-application when deciding which approach to take.

Consider the example of an application composed of many forms. In a modular application, a main application loads each form as a module with the `ModuleManager`. In an application that uses sub-applications, each form is a separate application that the `SWFLoader` loads into the main application. These two approaches are similar in many ways, but also have many differences.

Many of the reasons to use modules also apply to sub-applications. Main applications usually load sub-applications rather than embed their functionality. The result is generally a smaller initial download size and shorter load time for the main application. In addition, this promotes better encapsulation of related functionality, which can make development and maintenance easier.

Modules and sub-applications have the following similarities:

- Are compiled SWF files
- Can be loaded and unloaded at run time
- Promote encapsulation of related functionality
- Allow for an asynchronous development environment
- Can be recompiled without having to recompile the main application
- Support debugging
- Can be loaded locally or remotely (with the appropriate permissions)
- Are cached by the browser
- Can be preloaded
- Support progress events for getting the status while loading
- Can dynamically access methods and properties of the main application

Modules and sub-applications are also different. The main difference is that modules typically share classes with their host application. This sharing of classes creates a dependency between the module and the main application. Sub-applications, on the other hand, typically have their own versions of the classes, so less dependency exists between them and the main application.

Other differences between modules and sub-applications include the following:

File size Modules and their host applications are often smaller in file size because you can externalize shared classes. You can't always externalize shared classes for sub-applications. In cases where the versions of the applications are different, each application must have its own set of classes.

Reuse Modules are more tightly bound to the host application. They use interfaces to communicate with the loading application. This means that as your application changes, if your application uses modules, you'll probably need to recompile all of them if you move the main application to a later version of Flex.

Versioning Modules do not support multi-versioning. The main application and all modules must be compiled by the same version of the Flex framework.

Manager classes Modules and their host applications typically share manager classes. The modules are in a child application domain, whereas sub-applications often have their own instance of the manager class so that they can be multi-versioned.

ActionScript-only Modules can be either MXML-based or ActionScript-only; applications and sub-applications are not typically pure ActionScript.

Application domains sub-applications can be loaded into sibling application domains or child application domains, whereas modules must be loaded into a child application domain of the host application.

Security domains Modules must be loaded into the same security domain as their host application. Sub-applications can be loaded into either the same security domain or a different security domain.

For more information about modules, see *Creating Modular Applications*.

Comparing the SWFLoader and Loader controls

You can use the SWFLoader and Loader classes to load sub-applications into a main application. In most circumstances, use the SWFLoader class. This class wraps the Loader class and provides additional functionality that makes it easier to use for loading sub-applications into main applications.

The SWFLoader control has the following features:

- Supports Flex styles and effects; the Loader class does not have any inherent support for styles and effects.
- Lets you monitor the progress of a load inherently (if you use the Loader class, you have to first get a reference to a LoaderInfo object).
- Is a UIComponent. As a result, the SWFLoader control participates in the display list and adds children to the display list without having to write additional code.
- Resizes and scales the contents automatically.
- Does not require that the SWF file be an instance of the Application class, it just checks if an Application exists, and handles sizing differently.
- Can be multi-versioned. The Loader class does not have built-in support for multi-versioning.

Communicating across domains

Whenever you have a main application and sub-applications that are in different application domains, the recommended way to communicate between the applications is through event passing. If something that happens in the sub-application affects the main application, the sub-application triggers an event. This event can be caught in the main application. The event defines the properties necessary to respond to the event. For example, suppose a user moves the mouse pointer over a control in a sub-application that has a ToolTip. The sub-application's ToolTipManager creates the ToolTip and sends an event that requests the main application's ToolTipManager to display it. This interaction works across application domains that are within the same security domain.

Communication across security domains use the LoaderInfo.sharedEvents dispatcher and custom events. This process is largely transparent to you; you only need to know which events to register with event listeners. The LoaderInfo.sharedEvents class dispatches events in a sub-application that should be handled by a manager in the main application. The event contains data that define it, and that data is marshaled across the security domain. It is important to note that custom event classes cannot be strongly typed across the security domain by the receiver. Only classes defined by Flash Player can be strongly typed.

In many cases, communicating across security domains is transparent to the developer. The Flex framework handles the details of event passing and data marshaling for you. For example, when using functionality managed by the FocusManager or PopUpManager, you do not generally write additional code to make application interoperability work. The underlying Flex classes trigger the events and marshal the data across the domains so that the experience is seamless. In some cases, such as when you use custom classes to define objects, you must write custom code to marshal the data across the security domain.

Loading same-domain and cross-domain applications

The level of interaction that a main application and sub-application have depends on the level of trust between them. They can be trusted or untrusted. By default, applications that are loaded from a different web domain are untrusted, and applications that are loaded from the same web domain are trusted. You can, however, make a cross-domain application trusted.

Same-domain applications are applications that are loaded into the main application from the same web domain as the main application. By default, they are loaded into the same security domain and are trusted. Trusted applications can be loaded into the same application domain as the main application, which means they share their class definitions with the main application. They can also be loaded into a different, sibling application domain, which means that they contain their own definitions for all their classes and can be multi-versioned.

Cross-domain applications are applications that are loaded into a main application from a different web domain as the main application. For example, if the main application is located at `domainA.com`, and the sub-application is located at `domainB.com`, then the sub-application is a cross-domain application. Cross-domain applications are often known as untrusted applications, although you can specify that a remote application be trusted (except when using AIR).

When loading cross-domain SWF files, consider these points:

- The main application might have to call the `Security.allowDomain()` method.
- The sub-application's server might require a `crossdomain.xml` if you try to load data from the target application.
- The main application can define the level of trust between the main application and the sub-application by setting the value of the `trustContent` property (if the `crossdomain.xml` file permits it, and you are not using AIR).

Untrusted applications are loaded into a different security domain as the main application. As a result, Flash Player provides only limited interoperability between applications that do not trust each other because of the security concerns.

You can load the content of any SWF file from any web domain regardless of whether target web server has a policy file, but you need permissions to read the SWF file's data. To load a cross-domain application and access its data, the target server that hosts the remote sub-application must have a policy file called `crossdomain.xml` on it. This file must specifically allow access from the source server. In the previous example, if you wanted to access the loaded application's data, `domainB` must have a policy file on it that lets `domainA` load from it. Without a policy file, Flash Player throws a security error if it tries to access the application's data. A policy file must be in the root of the domain, or its location must be specified explicitly in the loading application. For more information about the `crossdomain.xml` file, see [Using cross-domain policy files](#).

Sub-domains that are not the same, such as `www1.domainA.com` and `www2.domainA.com`, also apply when deciding if a sub-application is cross-domain or not. In this example, if the main application is on `www1.domainA.com`, and the sub-application is on `www2.domainA.com`, then the sub-application is considered untrusted by default. To load data from the sub-application, the sub-application's domain would require a policy file.

You cannot test loading cross-domain applications directly in Flex Builder because the sub-applications must be loaded from a separate domain. As a result, to build the main application and the sub-application in Flex Builder, set up the compilation process to deploy the applications to separate servers for testing.

Cross-domain applications can be "import loaded," which means that they are loaded into the same security domain as the main application. You do this by setting the `trustContent` flag to `true` on the `SWFLoader`, or by specifying that the application load into the same security domain `domain` on the `LoaderContext`. You should only import load a cross-domain application if you know for sure that it can be trusted. For more information on using the `LoaderContext` to load sub-applications, see "Specifying a `LoaderContext`" on page 21.

Another way to load cross-domain applications into the same security domain is to use the `Security.allowDomain()` method. You first load the application as a sandboxed application (from a different web domain, without setting `trustContent` to `true`). In the main application, you call the `allowDomain()` method on the sub-application's domain in the `SWFLoader` control's `complete` event handler. In the sub-application, you call the `allowDomain()` method on the main application's domain. You must make this call early in the sub-application's lifecycle. For example, in the application's `preinitialize` event. If you use the `allowDomain()` method to load applications, you do not need a `crossdomain.xml` file on the target domain.

You can load a local (or same-domain) application into a different security domain. To do this, you can use a URL string for the SWFLoader's `source` property that is different from the path to the main application. For example, you could specify an IP address for the `source` property. When Flash Player compares the domain names, they are recognized as different, and the sub-application loads as an untrusted application. This is a common approach if you want the loaded application to be sandboxed, but deploy the applications into the same web domain. You could also create separate sub domains on the same server to ensure that the sub-application is loaded into a separate security domain.

sub-applications access remote data based on the way they were loaded. If you import load an application, the sub-application is effectively running from within the main application's security domain. As a result, it would now need permissions to access data on its domain of origin. If you load a sub-application into a separate security domain, it runs within its original domain and can access resources on that domain as normal.

Creating and loading sub-applications

You use the SWFLoader control to load sub-applications into your main Flex application. The default behavior of the SWFLoader control assumes that the application you are developing loads trusted sub-applications that were compiled with the same version of the Flex framework. These applications are typically loaded from the same web domain as the main application.

You can load the following other types of applications:

Sandboxed applications Sandboxed applications contain sub-applications that are loaded into separate security domains. As a result, they can be multi-versioned, but are untrusted. This is the recommended approach for any third-party application, or for many multi-versioned applications that use RPC classes or DataServices-related functionality. For more information, see "Developing sandboxed applications" on page 31.

Multi-versioned applications Multi-versioned applications are typically very large applications that load trusted sub-applications. The sub-applications might or might not have been compiled with the same version of the Flex framework as the main application. For more information, see "Developing multi-versioned applications" on page 39.

When developing large, single-versioned applications, you might consider using modules instead of sub-applications. For more information, see "Comparing loaded applications to modules" on page 13.

Sub-applications can run independently of the main application and other sub-applications. There should be no dependencies on the main application or other sub-applications for a sub-application to run.

When developing sub-applications in Flex Builder, you cannot just compile the main application and expect the sub-application to also recompile. You must compile them separately. Using Ant or some other automated build process can help. This process is different from developing modules, where if you compile the application that loads a module, Flex Builder also compiles the module.

Loading applications with the SWFLoader control

The SWFLoader control lets you load one Flex application into another Flex application. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the SWFLoader control. The SWFLoader control can also load content on demand programmatically, and monitor the progress of a load operation.

To load a SWF file into a SWFLoader control, you set the value of the SWFLoader control's `source` property. This property defines the location of the sub-application's SWF file. After you set the value of the `source` property, Flex imports the specified SWF file and runs it. For simple examples of using the SWFLoader tag in a Flex application, see [Creating a SWFLoader control](#).

In addition to SWF files, the SWFLoader control can also load several types of images (such as JPG, PNG, and GIF files) as SWF files. You cannot load a module, RSL, or style module with the SWFLoader control.

The default behavior for the SWFLoader control depends on the location of the loaded SWF file. If the loaded application is loaded locally, the default is to load the SWF file into the same security domain, and a child application domain. This loads an application that is trusted but single-versioned.

If the loaded application is loaded from a different server or in a separate sub domain than the main application, then the default is to load the application into a separate security domain and a separate, sibling application domain. The result is a multi-versioned application, but one that has some restrictions on it because it is untrusted by default. This is the recommended approach for all third-party applications, or any multi-versioned applications that use RPC classes or DataServices-related functionality.

When loading any application, you can explicitly set the values of properties on the SWFLoader control that set the application domain and the security domain. These properties define the trust level between the main application and the sub-application, and determine whether the applications are multi-versioned. The following table describes these properties of the SWFLoader control.

Property	Description
loadForCompatibility	<p>Determines if the loaded SWF file is a multi-versioned application.</p> <p>Set this property to <code>true</code> to indicate that the loaded application might be compiled with a different version of the Flex framework and you want your application to be able to interact with it. Setting the value of the <code>loadForCompatibility</code> property to <code>true</code> also causes the <code>LoaderContext</code> of the <code>Loader</code> to load the sub-application into a sibling application domain of the main application, rather than a child application domain.</p> <p>Set this property to <code>false</code> to disallow any version compatibility. If the loaded application was compiled with a different version of the framework, then the application will likely experience errors at run time (if the two applications use a shared resource whose API has changed between versions of the Flex framework). If the application was compiled with the same version of the framework, then it will load and interact with the main application as normal.</p> <p>The value of the <code>loadForCompatibility</code> property is ignored if you explicitly set the value of the <code>loaderContext</code> property.</p> <p>Setting this property has no impact on the sub-application's security domain.</p> <p>The default value of the <code>loadForCompatibility</code> property is <code>false</code>.</p> <p>If you set <code>loadForCompatibility</code> to <code>true</code> when developing your Flex applications, use this method of loading consistently in the future. If you change an application from one that is multi-versioned to one that is single-versioned, or vice versa, you should not expect the application to work the same way.</p> <p>For more information, see <code>SWFLoader.loadForCompatibility</code>.</p>
loaderContext	<p>Defines the context into which the child SWF file is loaded. The context defines the application domain and the security domain of the application.</p> <p>With the <code>loaderContext</code> property, you can set the security domain into which the application is loaded. You can also require that a policy file is in place before the SWF file is loaded. The default for a local application is to load the sub-application into the same security domain as the main application. For a remote (or cross-domain) application, the default is to load the sub-application into a different security domain.</p> <p>With the <code>loaderContext</code> property, you can also specify the application domain for the context. The default is a child of the <code>Loader</code>'s application domain. If you want to load a multi-versioned application, change this to a sibling application domain. In general, use the <code>loadForCompatibility</code> property rather than the <code>loaderContext</code> property to specify the application domain.</p> <p>If you explicitly set the value of the <code>loaderContext</code> property, then the <code>SWFLoader</code> control ignores the value of its <code>loadForCompatibility</code> and <code>trustContent</code> properties.</p> <p>The default value of the <code>loaderContext</code> property is <code>null</code>.</p> <p>You can only specify a <code>LoaderContext</code> in ActionScript. There are some restrictions on using the <code>loaderContext</code> property. For more information, see "Specifying a <code>LoaderContext</code>" on page 21.</p> <p>For more information, see <code>SWFLoader.loaderContext</code>.</p>

Property	Description
trustContent	<p>Determines whether the SWF file is loaded into the main application's security domain.</p> <p>Set to <code>true</code> to load the SWF file into the main application's security domain. Regardless of whether the SWF file is same-domain or cross-domain, the SWF file is now considered trusted (assuming that the policy file permits it to be). This means that it can access properties and methods of the main application, and vice versa. You should not set this to <code>true</code> for any third-party application unless you can absolutely trust the source of that application.</p> <p>Set this property to <code>false</code> to load a cross-domain SWF file into a separate security domain. It is now considered untrusted. The SWF file cannot access properties and methods of the main application, nor can the main application access the SWF file's properties and methods.</p> <p>The value of the <code>trustContent</code> property is ignored if you explicitly set the value of the <code>loaderContext</code> property.</p> <p>The default value of the <code>trustContent</code> property is <code>false</code>.</p> <p>For more information, see <code>SWFLoader.trustContent</code>.</p>

Specifying a LoaderContext

Specifying a loader context for the sub-application gives you control over the security domain and application domain that the application is loaded into. You specify the loader context by setting the values of the `securityDomain` and `applicationDomain` properties on the `LoaderContext` object.

You can only specify a loader context in ActionScript. You cannot set its value as an attribute in the `<mx:SWFLoader>` tag.

The following example assigns a custom `LoaderContext` to the `SWFLoader`. It defines the current security domain and a sibling application domain for the loader. This is only applicable to sub-applications that are single-versioned. For multi-versioned applications or sandboxed applications, you should use calls to the `allowDomain()` method to establish trust between the main application and the sub-application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppCustomLoaderContext.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="initApp()">
  <mx:Script>
    <![CDATA[
      import flash.system.SecurityDomain;
      import flash.system.ApplicationDomain;

      private function initApp():void {
        var context:LoaderContext = new LoaderContext();

        /* Specify the current application's security domain. */
        context.securityDomain = SecurityDomain.currentDomain;

        /* Specify a new ApplicationDomain, which loads the sub-app into a
           peer ApplicationDomain. */
        context.applicationDomain = new ApplicationDomain();

        contentLoader.loaderContext = context;
        contentLoader.source = "http://yourdomain.com/SubApp3.swf";
      }
    ]]>
  </mx:Script>
  <mx:SWFLoader id="contentLoader"/>
</mx:Application>
```

This example triggers the `initApp()` method on the application's `creationComplete` event. If you had instead set the `source` property of the SWFLoader control in MXML, you would use the `preinitialize` event to set the loader context. Waiting for the application's `creationComplete` event to set the loader context would occur too late in the application startup.

Setting the `context.securityDomain` property to the `currentDomain` makes the SWF file trusted by loading it into the same security domain as the main application. To do this, though, the sub-application's server must have a policy file that allows you to load it. The result is that the SWF file becomes trusted, and acts as if it was loaded locally. In AIR, you cannot specify any value for the `context.securityDomain` property.

You can only specify the value `SecurityDomain.currentDomain` for the `context.securityDomain` property. Attempting to pass any other security domain results in a `SecurityError` exception. If you do not specify any value for the `currentDomain` property, then a remote SWF file is loaded into its own security domain. This occurs unless you set the security domain in some other way, such as setting the value of the SWFLoader's `trustContent` property to `true`.

When specifying an application domain for the `context.applicationDomain` property, you can add a sub-application to a sibling application domain, a child application domain, or the same application domain as the loading application. The following table describes each of these methods.

Value of <code>context.applicationDomain</code>	Result
<code>new ApplicationDomain()</code>	<p>Loads the sub-application into a sibling application domain that is rooted on the same domain as the loading application. If the loading application is the top-level application, then both the loading application and the sub-application's application domain will be children of the system domain.</p> <p>You do this if you want to use compatibility mode, because applications that are in sibling application domains can each have their own class definitions. As a result, they can be compiled by different versions of the Flex framework.</p>
<code>new ApplicationDomain(ApplicationDomain.currentDomain)</code>	<p>Loads the sub-application into an application domain that is a child of the main application's application domain. All class conflicts are resolved when the application is loaded; the first class definition wins.</p> <p>Do not use this method if you want to load a multi-versioned application. That is because applications that are loaded into the same application domains must be compiled by the same version of the Flex framework.</p>
<code>application.currentDomain</code>	<p>Loads the sub-application into the same application domain as the main application. You typically do not use this setting when loading sub-applications. It is generally only used for RSLs and other specially compiled resources.</p>

If you specify a loader context when loading a sub-application, do not load more than one application into the same application domain. In other words, do not use the same loader context for more than one sub-application.

Unloading applications with the SWFLoader control

To unload a sub-application that you loaded with the SWFLoader control, call the SWFLoader control's `unloadAndStop()` method, as the following example shows:

```
myLoader.unloadAndStop(true);
```

You can also set the SWFLoader's `source` property to `null`. This results in a call to the `SWFLoaderObject.content.loaderInfo.loader.unload()` method. You can call this method explicitly, but only for trusted applications.

To free up the memory that was used by the loaded sub-application, ensure that no references to objects or classes in the sub-application exist. The `unload()` method frees the loader's reference to the sub-application's bytes, but if code in the sub-application is still in use, then it is not garbage collected.

Flash Player also unloads a sub-application when the same SWFLoader control loads a new sub-application. The original content is removed before the new application is loaded.

If the sub-application contains any user interface classes that are not in the main application, the styles for those classes are loaded into the main application's StyleManager. The sub-application will not free up the sub-application's memory. In this case, load styles before loading the sub-application by using compiler options or loading run-time style sheets.

Accessing the main application from sub-applications

There are several ways to access the methods and properties of the main application from the sub-application. These ways only work for sub-applications that are loaded as children into a main application's application domain. You cannot use these for sandboxed applications, or for multi-versioned applications.

These methods include:

- Using the `application` property of the Application class. This property accesses the root application from anywhere in the application. For more information, see [Using the mx.core.Application.application property](#).
- Using the `parentDocument` property of the Application class. This is useful if you have multiple applications embedded and want to just access the immediate parent. For more information, see [Using the parentDocument property](#).

Be aware that you might encounter security errors if you try to access members that should not be accessed.

Accessing sub-applications from the main application

Although you can access the sub-application from the main application, you cannot directly reference methods and properties. This is because the members of the sub-application, unless it is embedded at compile-time, are not known by the compiler when the main application compiles.

You can, however, get a reference to the sub-application's SystemManager. You can then treat members of the sub-application as dynamic properties and methods of the sub-application's object.

The following example loads a remote sub-application into the SWFLoader. It sets the value of the `trustContent` property to `true` so that the sub-application is loaded into the same security domain as the main application. It casts the `content` property of the SWFLoader to a SystemManager so that the application's members can be accessed dynamically. It then calls a method and accesses a property of the sub-application.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppUsingSubAppMembers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.managers.SystemManager;

      private function getValueFromSubApp():void {
        /* Cast the SWFLoader's content as a SystemManager
           to access the sub-application. */
        var subApp:SubApp2 =
          (contentLoader.content as SystemManager).application as SubApp2;

        /* Call a method and access a property of the
           sub-application. */
        label1.text = subApp.doSomething() + subApp.answer;
      }
    ]]>
  </mx:Script>
  <mx:SWFLoader id="contentLoader" visible="false"
    height="0"
    width="0"
    trustContent="true"
    source="http://yourdomain.com/SubApp2.swf"
  />
  <mx:Panel id="myPanel2"
    paddingLeft="10"
    paddingBottom="10"
    paddingTop="10"
    paddingRight="10"
  >
    <mx:Label id="label1"/>
    <mx:Button id="b2" label="Call SubApp2" click="getValueFromSubApp()"/>
  </mx:Panel>
</mx:Application>
```

The following example shows the sub-application that is loaded by the main application in the previous example. It defines a public property, `answer`, as well as a public method, `doSomething()`, that returns a `String`.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/SubApp2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      // Define a public property.
      public var answer:int = 42;

      // Define a public method that returns a String.
      public function doSomething():String {
        return "The answer is ";
      }
    ]]>
  </mx:Script>
</mx:Application>
```

For more information about accessing sub-applications from the main application, see `SWFLoader` control.

If the sub-application is loaded remotely, you can access its members only if it is in the same security domain as the main application, or if you and the sub-application call the `Security.allowDomain()` method. In this case, you must call the `Security.allowDomain()` method from the main application on the sub-application's domain, and the sub-application must call this method on the main application's domain. You must also call the `allowDomain()` method early in the sub-application's lifecycle. For example, call it in a `preinitialize` event handler.

Creating class instances from loaded applications

You can create instances in a main application of classes that are defined in a loaded sub-application. You can then add these objects to your main application and interact with them as you would interact with any other object in the display list. To access class definitions in a sub-application, you get the definitions from the sub-application's application domain.

Each application domain contains definitions of all the classes within it. There is an `ApplicationDomain` object that you access with a reference to the loaded application's `LoaderContext`. The `ApplicationDomain` object has two methods, `hasDefinition()` and `getDefinition()`. The `hasDefinition()` method lets you detect if a class definition exists. If a class definition exists, the `getDefinition()` method lets you create an instance of that class in your main application.

You can't add a `UIComponent` that is defined in another application domain to the main application's display list. As a result, to create an instance of a class that is defined in another application domain, the sub-application must be loaded into a child application domain of the main application's application domain (you cannot set the value of the `SWFLoader`'s `loadForCompatibility` property to `true`). The sub-application must also be in the same security domain as the main application (`trustContent` must be `true`).

You can only create the instance of the class dynamically when the class is defined in an application that is loaded at run time. This is because the compiler does not have access to classes in loaded applications at compile time, so it cannot check linkages. If the sub-application was embedded at compile time, then you would not have to create the instance dynamically, the class definition would be available to the compiler. In this case, you could instead use the `new` keyword with the specific class type rather than a generic `Class` type.

The following example main application loads a sub-application with the `SWFLoader` control. It sets the value of the `trustContent` property to `true` and defines a method, `createClassInstance()`. This method gets the definition of the custom class from its loaded application's application domain. The example then creates an instance of this class and sets a property on it before adding it to the display list.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainAppUsingSubAppDefinitions.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.core.UIComponent;
      public function createClassInstance():void {
        // Check that the definition exists.
        if (contentLoader.loaderContext.applicationDomain.hasDefinition('MyButton')) {
          var objClass:Class =
contentLoader.loaderContext.applicationDomain.getDefinition('MyButton') as Class;
          if (objClass != null) {
            var newObject:UIComponent = UIComponent(new objClass());

            // Set properties on the custom class as an associative array.
            newObject["label"] = "Click Me";

            // Add the new instance to the second panel in this application.
            myPanel2.addChild(newObject);
          }
        }
      }
    ]]>
  </mx:Script>
  <!-- The SWFLoader in the first panel loads the sub-application that contains a definition
of MyButton. -->
  <mx:Panel id="myPanel" title="SubApp1 Loaded by main application">
    <mx:SWFLoader id="contentLoader" trustContent="true" source="SubApp1.swf"/>
  </mx:Panel>

  <!-- This application adds an instance of the MyButton class to the second panel after
the content is loaded. -->
  <mx:Panel id="myPanel2" title="Instance of a Class Defined By SubApp1"/>

</mx:Application>

```

In this case, the `hasDefinition()` method returns a boolean to tell if a class is present in the loaded SWF. If it is present, the sub-application can obtain the `Class` using the `getDefinition()` method. You can then create instances by using the `new` keyword with the returned class.

Also notice that the `createClassInstance()` method is public. If it were private, the sub-application would not be able to call it.

The sub-application, `SubApp1.swf`, statically links a custom class called `MyButton`. It also calls the parent application's method when it is done. You cannot call that method in the parent application's `applicationComplete` event, because that event will likely be triggered too early in the initialization process. Wait for the sub-application to complete its loading and initialization before you create an instance of a class that is defined in it.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/SubApp1.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:custom="*"
  layout="vertical"
  creationComplete="initApp()"
  applicationComplete="mx.core.Application.application.createClassInstance()"
>
  <mx:Script>
    <![CDATA[
      private function initApp():void {
        var child:DisplayObject = getChildAt(0);
        var childClassName:String = getQualifiedClassName(child);

        // Show that the qualified class name of the custom button is MyButton.
        trace(childClassName);
      }
    ]]>
  </mx:Script>
  <custom:MyButton id="myButtonId" label="Click Me"/>
</mx:Application>

```

The `MyButton` class is a simple MXML component that extends `Button` and defines its color as red. The following example shows this custom class:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MyButton.mxml -->
<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml" color="red">
</mx:Button>

```

Listening for mouse events with loaded applications

It is possible to listen for mouse events in a main application from a loaded sub-application that is in a child application domain. To listen for these events, you listen for events such as `MOUSE_UP` and `MOUSE_MOVE` on the sub-application's `systemManager.topLevelSystemManager`. When the sub-application is in a child application domain, the `topLevelSystemManager` property refers to the main application's `SystemManager`.

The following application shows how to access mouse events on the `topLevelSystemManager` in a sub-application that was loaded into a child application domain. If the sub-application is not in a child application domain, then you must handle access to the `SystemManagers` differently. For more information, see the examples in “Listening for mouse events with sandboxed applications” on page 35 and “Listening for mouse events in multi-versioned applications” on page 42.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="setup()"
    height="250"
>
    <mx:Script>
    <![CDATA[
        import mx.core.UIComponent;
        import mx.managers.PopUpManager;

        [Bindable]
        public var data:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];

        public var zoomTool:UIComponent;

        public function setup():void {
            // Draw the zoom rectangle.
            zoomWidget.graphics.lineStyle(1);
            zoomWidget.graphics.beginFill(0, 0);
            zoomWidget.graphics.drawRect(0, 0, 17, 17);
            zoomWidget.graphics.endFill();

            // Listen for mouse down events.
            zoomWidget.addEventListener(MouseEvent.CLICK, zoom_mouseDownHandler);
        }

        private var lastX:int;
        private var lastY:int;

        private function zoom_mouseDownHandler(event:MouseEvent):void {
            // When the mouse is down, listen for the move and up events.
            systemManager.topLevelSystemManager.addEventListener(
                MouseEvent.CLICK, zoom_mouseMoveHandler, true);
            systemManager.topLevelSystemManager.addEventListener(
                MouseEvent.CLICK, zoom_mouseUpHandler, true);

            // Update the last position of the mouse.
            lastX = event.stageX;
            lastY = event.stageY;

            // Create and pop up the zoomTool. This is what is dragged around.
            // It must be a popup so that it can float over other content.
            zoomTool = new UIComponent();
            PopUpManager.addPopUp(zoomTool, this);
            var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
                zoomWidget.transform.pixelBounds.y);
            pt = zoomTool.parent.globalToLocal(pt);
            zoomTool.x = pt.x;
            zoomTool.y = pt.y;
            zoomTool.graphics.lineStyle(1);
            zoomTool.graphics.beginFill(0, 0);
            zoomTool.graphics.drawRect(0, 0, 17, 17);
            zoomTool.graphics.endFill();

            // Hide the rectangle that was the target.
        }
    ]]>
</mx:Script>
</mx:Application>

```

```

        zoomWidget.visible = false;
    }

    private function zoom_mouseMoveHandler(event:MouseEvent):void {
        // Update the position of the dragged rectangle.
        zoomTool.x += event.stageX - lastX;
        zoomTool.y += event.stageY - lastY;
        lastX = event.stageX;
        lastY = event.stageY;

        var bm:BitmapData = new BitmapData(16, 16);

        // Capture the bits on the screen.
        bm.draw(DisplayObject(systemManager.topLevelSystemManager), new
            Matrix(1, 0, 0, 1, -zoomTool.transform.pixelBounds.x - 2,
                -zoomTool.transform.pixelBounds.y - 2));

        // Create a Bitmap to hold the bits.
        if (zoomed.numChildren == 0) {
            var bmp:Bitmap = new Bitmap();
            zoomed.addChild(bmp);
        } else
            bmp = zoomed.getChildAt(0) as Bitmap;

        // Set the bits.
        bmp.bitmapData = bm;

        // Zoom in on the bits.
        bmp.scaleX = bmp.scaleY = 8;
    }

    private function zoom_mouseUpHandler(event:Event):void {
        // Remove the listeners.
        systemManager.topLevelSystemManager.removeEventListener(
            MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
        systemManager.topLevelSystemManager.removeEventListener(
            MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);

        // Replace the target rectangle.
        zoomWidget.visible = true;

        // Remove the dragged rectangle.
        PopUpManager.removePopUp(zoomTool);
    }

```

```

    }

    ]]>
</mx:Script>
<mx:HBox>
  <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
    <mx:Label text="Drag Rectangle"/>
    <mx:UIComponent id="zoomWidget" width="17" height="17"/>
    <mx:Canvas id="zoom"
      borderStyle="solid"
      borderThickness="2"
      width="132"
      height="132"
    >
      <mx:UIComponent id="zoomed" width="128" height="128"/>
    </mx:Canvas>
  </mx:HBox>
  <mx>List dataProvider="{data1}"/>
</mx:HBox>
</mx:Application>

```

The following example main application loads the previous example application. It uses the default settings to load the sub-application.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern2.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" backgroundColor="#FFFFFF">
  <mx:Text text="Default (trusted application in child ApplicationDomain):"/>
  <mx:SWFLoader id="swf1" source="ZoomerPattern2.swf"/>
</mx:Application>

```

In this example, you cannot drag the rectangle outside the sub-application's boundaries, and the `mouseUp` event does not get triggered.

This example application uses the `systemManager.topLevelSystemManager` property to get a reference to the main application's `SystemManager`. If the application were a stand-alone application, you could use the `systemManager` property to register event listeners, as the following example shows:

```

systemManager.addEventListener(MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
systemManager.addEventListener(MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);

```

This works when the application is a stand-alone application, but not when the application is loaded as a child. The top-level `SystemManager`, which is not the sub-application's `SystemManager`, parents the `zoomTool` class.

Embedded fonts in loaded applications

Flex applications and modules in the same application domain are able to use the same embedded fonts by specifying the font name. For example, a font embedded in the main application can be used by a sub-application, as long as the applications are in the same application domain.

Models and singletons in loaded applications

Models that are implemented as singletons, and other singletons that the main application and its sub-applications share, do not work if the sub-application and main application are in separate application domains.

To share models and other singletons, you can write your own marshaling code or create a bootstrap loader that stores definitions of those classes.

Adobe recommends not sharing data models and other singletons with untrusted applications.

For more information, see “Bootstrap loading” on page 45.

Optimizing loaded applications

When creating main applications and sub-applications, try to have as little overlap in class definitions as possible. In other words, try to have only one definition of each class so that the aggregate size of the applications is as small as possible.

SWF files can be large, especially if they contain embedded assets or use many class libraries that are linked into them. If you are sure that the main application and the loaded application will always be compiled with the same version of the Flex framework, then you can externalize overlapping assets by using RSLs or other techniques with the compiler.

In general sub-applications must load their own RSLs. You typically do not compile a sub-application against the main application’s RSL or another sub-application’s RSL, or even if you do, the sub-application will load the RSL on its own when it starts. The result is that the RSL is loaded twice, which likely negates the benefits of using RSLs.

When you develop sandboxed or multi-versioned applications, you cannot always externalize overlapping class definitions, because the class definitions might be different. For example, if two applications are compiled with different versions of the Flex framework, then each application must have its own definition of the manager classes (such as the `LayoutManager` or the `CursorManager`) and other classes in the framework. As a result, do not externalize those classes by compiling the applications against the same RSLs.

In previous versions of Flex, you were required to initialize certain manager classes such as the `FocusManager` if you loaded a module or sub-application into a main application that did not use those manager classes. You are no longer required to do this if you load the sub-application into a sibling application domain where each application has its own class definitions. However, each security domain must include a definition of the `PopUpManager` class in case an untrusted sub-application displays a modal dialog box.

For information on externalizing overlapping assets, see [Externalizing application classes](#).

Developing sandboxed applications

Sandboxed applications contain sub-applications that are loaded into separate security domains. By definition, they are therefor loaded into separate application domains as well. As a result, they can be multi-versioned, but are untrusted. Because they are untrusted, their interoperability with the main applications is limited. Types of sandboxed applications include portals, mashups, and dashboards.

If you are using any third-party applications, you should load them as sandboxed applications. In addition, if you are using multi-versioned applications that use RPC classes or `DataServices`-related functionality, you should also consider loading them as sandboxed applications. Otherwise, you might be required to provide additional code such as a bootstrap loader.

In a sandboxed configuration, loaded sub-applications are typically not on the same domain as the main application. This means that the applications might not necessarily trust the loaded applications by default. In addition, all sub-applications are not necessarily always visible at the same time, so the main sandboxed application must be able to load and unload sub-applications at any time.

In sandboxed applications, each sub-application is loaded into a separate application domain and a separate security domain. The interoperability across security domains is very limited. The sub-application cannot access most stage properties, methods, and events. It cannot get mouse and keyboard events from other security domains. It also cannot perform drag and drop operations to or from the main application, and pop-up controls are clipped at the boundaries of the sub-application. Data sharing between the main application and sub-application requires marshaling.

If you try to use the parent chain of an application object to access properties of the main application from a sub-application in a different security domain, you will encounter security errors at run-time. In addition, you cannot access the application through the `SWFLoader.content` object.

For details about the architecture of a sandboxed application, see “About sandboxed applications” on page 5.

Pop-up controls in sandboxed applications

Pop-up controls are parented by the application at the sandbox root of their security domain. This is because the sandbox root handles requests to display a modal window from its children.

Because pop-up controls are parented by the sandbox root, centering a popup in a sandboxed application centers it in the area of the screen occupied by the sub-application and not the entire application. It also means that pop-up controls are sometimes clipped by scroll bars and masks on the sub-application.

A sub-application in a separate security domain from the main application has the following behavior:

- Launching a modal dialog box dims the entire application, but the pop-up can only be dragged within the boundaries of the sub-application.
- Centering a pop-up centers it over the sub-application, not the main application.
- Dragging pop-up controls works over the sub-application only. If you drag a pop-up outside the sub-application, it is clipped.
- Focus shifts to the pop-up control when you first launch a pop-up.

A sandboxed application cannot display a window or dialog box outside the bounds of its application. This rule prevents an untrusted application from phishing for passwords by displaying a dialog box on top of all the applications. When displaying a popup window, the `PopUpManager` checks if the parent application trusts it and if it trusts the parent application before asking the parent to host the window. If the parent hosts a window it is displayed over the parent's content as well as the child's content. If no mutual trust exists between a main and sub-application, then the `PopUpManager` hosts the dialog box locally so that it can only be displayed over the content of the application itself. But if the parent trusts the child, the dialog box is not clipped by the boundaries of the child's application.

When a main application does not trust a sub-application, the main application's `SWFLoader` uses masking with a `scrollRect` and scroll bars to keep the sandboxed application's content restricted to its own application space.

Pop-up-related controls such as `ColorPicker`, `ComboBox`, `DateField`, `PopUpButton`, `PopUpMenuButton`, and `Menu` sometimes display their contents in unexpected ways if their normal position would cause them to be clipped.

Alert controls in sandboxed applications

Alerts, like other pop-up controls in sandboxed applications, are clipped at the edge of the loaded application. When an Alert is being displayed, the main application and all sub-applications are covered with a modal dialog box to prevent interaction with their controls. The blur effect only applies to the sub-application that launched the Alert box, and its child applications. The blur effect is not applied to the parent application or sibling applications.

Styles and style modules in sandboxed applications

The StyleManager does not pass styles from a parent application to a child in a different application domain or security domain. Therefore, define styles within your sub-application and do not depend on the sub-application inheriting styles from the main application. Similarly, a main application does not inherit styles from a sub-application.

If you want a main application and a sub-application to use the same runtime style sheets, load the style module into both the main application and the sub-application. Sub-applications do not inherit styles defined in style modules that are loaded into main applications. Similarly, main applications do not inherit styles that are defined in style modules that are loaded into sub-applications.

A style module must be compiled with the same version of the Flex framework as the application into which it is loaded. The main application and sub-application might not be able to load the same style module, unless they are compiled with the same version of the framework.

When loading a style module into a sub-application, if you don't specify an application domain, the module is loaded into a *sibling* application domain of the sub-application. This can result in an error when the sub-application tries to use classes that are defined in the style module. To load a style module into a sub-application and use its styles, load the style module into a *child* application domain of the sub-application. The `loadStyleDeclarations()` method of StyleManager has two optional parameters, `applicationDomain` and `securityDomain`. You use these properties to control the application domain and the security domain into which style modules get loaded.

The following example loads a style module into a child application domain of the sub-application:

```
private function loadStyle():void {
    /* Load style module into a child ApplicationDomain by specifying
       ApplicationDomain.currentDomain. */
    var eventDispatcher:IEventDispatcher = StyleManager.loadStyleDeclarations(
        currentTheme + ".swf", true, false, ApplicationDomain.currentDomain);
    eventDispatcher.addEventListener(StyleEvent.COMPLETE, completeHandler);
}
```

For more information on using style modules, see [Loading style sheets at run time](#).

Fonts in sandboxed applications

Applications that are in the same application domain are able to use the same embedded fonts by specifying the font name. However, if the sub-application is loaded into a different application domain (as is the case with sandboxed applications), then the sub-application must embed the font to use it.

Focus in sandboxed applications

The FocusManager class in the main application and sub-application integrate to create a seamless focus scheme. Users can “tab through” the sub-application, regardless of whether the application is in the same or a different security domain as the main application. Shift tabbing also works. The FocusManager class is one of the few manager classes that supports interoperability even across security domains.

When an application is loaded, the main application keeps track of that SWF file in a list of focus candidates. When the user moves focus into the sub-application, the sub-application's FocusManager takes over focus duties until the user moves focus outside the sub-application. At that time, the main application's FocusManager resumes control.

When a pop-up is dismissed, the focus is moved to the last place that had focus. This behavior can be another pop-up or it can be in the main application.

When focus is on a control that is in a different security sandbox, calls to the `getFocus()` method on that application's FocusManager return `null`. Calls to the `UIComponent.getFocus()` method also return `null`.

The FocusManager's `moveFocus()` method lets you programmatically transfer focus to a control under the jurisdiction of another FocusManager. It also lets you transfer the control of focus to another FocusManager.

Focus management across application domains works even with modal dialog boxes. When a different top-level window is activated, the SystemManager deactivates the FocusManager in the formerly active top-level window. The SystemManager also activates the FocusManager in the other window and changes the depth level (z-order) of the windows.

Cursors in sandboxed applications

If the applications are in different security domains, then a custom cursor in the sub-application only appears over the area of the screen that is allocated to that sub-application. Moving the mouse outside the bounds of the sub-application passes control of the cursor to the main application's CursorManager.

These rules apply to the busy cursor as well. If a busy cursor is visible and you move the cursor to the main application that is in a different security domain, the cursor changes back to the last used cursor in the main application.

Localizing sandboxed applications

As with style modules, the main application cannot access resource modules used in a sub-application, and vice versa. Each sub-application must load its own resource modules.

Each multi-versioned application has its own ResourceManager instance. As a result, each sub-application has its own localeChain.

If more than one sub-application has resource bundles for the same locale with the same name, then the first one in wins. The contents of all resource bundles of that name in other sub-applications are ignored. Those sub-applications use the one that was defined first.

Like style modules, load resource modules into the child application domain of a sub-application. You control the application domain and the security domain into which resource bundles are loaded. The `loadResourceModule()` method of IResourceManager has two optional parameters, `applicationDomain` and `securityDomain`.

Also like style modules, all resource modules in an application must be compiled with the same version of the Flex framework. Do this whether that application is a main application or a sub-application. You cannot use multiple resource modules that were compiled with two different versions of the framework in the same main application or sub-application.

The following example loads a resource module into a child application domain of the sub-application:

```
private function loadBundle():void {
    /* Load resource module into a child ApplicationDomain by specifying
       ApplicationDomain.currentDomain. */
    var eventDispatcher:IEventDispatcher = ResourceManager.loadResourceModule(
        "MyBundle.swf", true, false, ApplicationDomain.currentDomain);
    eventDispatcher.addEventListener(StyleEvent.COMPLETE, completeHandler);
}
```

ToolTip objects in sandboxed applications

When in a different security domain, ToolTip objects are parented by the sub-application's SystemManager and are therefore clipped and masked by the main application. The ToolTipManager styles and positions the tip in a sub-application so that it fits within the sub-application's area of the screen only. If the ToolTip object is larger than the area of the sub-application, the ToolTip object is clipped.

ToolTip styles in the main application are not inherited by ToolTip objects in the sub-application.

This applies to error tips and data tips on List objects in sandboxed sub-applications as well.

Layouts in sandboxed applications

For controls that have pop-up or drop-down menus, when they are in a separate security domain, they are initially displayed unclipped. These controls are restricted to the sub-application's space, so if they try to go outside that bounding area, they are clipped.

Deep linking in sandboxed applications

The `BrowserManager` controls deep linking support in Flex applications. It is a singleton within its security domain. Sub-applications in sibling application domains cannot access the main application's `BrowserManager`, regardless of whether a sub-application is trusted or untrusted. As a result, a sub-application in a separate security domain or application domain cannot modify the URL nor can it access the URL.

If the sub-application is untrusted, do not give it access to the URL. If the sub-application is trusted (such as with a multi-versioned application that is not sandboxed), you can write custom code that handles the interaction between the sub-application and the main application's `BrowserManager`.

If you try to get an instance of the `BrowserManager` from within a sub-application, Flash Player throws an error.

Dragging and dropping in sandboxed applications

When the sub-application is in a different security domain from the main application, the user cannot drag data between the applications. The `DragProxy` lets the user drag an item to the edge of the sub-application's area of the screen. At that point, the mouse cursor changes back to whatever mouse cursor is correct for the new security domain.

The proxy for the item stays at the boundary of the sub-application, and might be clipped.

Listening for mouse events with sandboxed applications

Mouse interaction between sub-applications and main applications can be confusing, especially when those events occur in different application domains. This interaction is further muddied when the applications are in different security domains. To listen for mouse events outside the security domain, you use the `SandboxMouseEvent` object. You can listen for this event in a main application for a mouse event that is triggered from the sub-application, and vice versa.

The following application is like the application in the topic, "Listening for mouse events in multi-versioned applications" on page 42, with some exceptions. For example, while the `MouseEvent.MOUSE_MOVE` and `MouseEvent.MOUSE_UP` events are registered, the `SandboxMouseEvent.MOUSE_UP` event is also registered. This event can be registered by any `SystemManager` within the security domain. All applications can then receive notification if this event is triggered. To get the mouse position, this application uses the `globalToLocal()` method. You can see how to determine the absolute position of an object in a sub-application when you don't have access to the stage.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern4.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="setup()"
    height="250"
>
    <mx:Script>
    <![CDATA[
        import mx.core.UIComponent;
        import mx.events.SandboxMouseEvent;
        import mx.managers.PopUpManager;

        [Bindable]
        public var data:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];

        public var zoomTool:UIComponent;

        public function setup():void {
            // Draw the zoom rectangle.
            zoomWidget.graphics.lineStyle(1);
            zoomWidget.graphics.beginFill(0, 0);
            zoomWidget.graphics.drawRect(0, 0, 17, 17);
            zoomWidget.graphics.endFill();

            // Listen for mouse down events.
            zoomWidget.addEventListener(MouseEvent.CLICK, zoom_mouseDownHandler);
        }

        private var lastX:int;
        private var lastY:int;

        private function zoom_mouseDownHandler(event:MouseEvent):void {
            // When the mouse is down, listen for the move and up events.
            // The getSandboxRoot() method lets you listen to all mouse activity in your
            // SecurityDomain.

            systemManager.getSandboxRoot().addEventListener(
                MouseEvent.CLICK, zoom_mouseClickHandler, true);
            systemManager.getSandboxRoot().addEventListener(
                MouseEvent.CLICK, zoom_mouseUpHandler, true);

            // The SandboxMouseEvents provide you with some mouse information,
            // but not its position
            systemManager.getSandboxRoot().addEventListener(
                SandboxMouseEvent.CLICK, zoom_mouseUpHandler);

            // Update last position of the mouse.
            lastX = event.stageX;
            lastY = event.stageY;

            // Create and pop up the zoomTool. This is the rectangle that is dragged around.
            // It must be a popup so that it can float over other content.
            zoomTool = new UIComponent();
            PopUpManager.addPopUp(zoomTool, this);
        }
    ]]>
    </mx:Script>
</mx:Application>
```

```

        var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
            zoomWidget.transform.pixelBounds.y);
        pt = zoomTool.parent.globalToLocal(pt);
        zoomTool.x = pt.x;
        zoomTool.y = pt.y;
        zoomTool.graphics.lineStyle(1);
        zoomTool.graphics.beginFill(0, 0);
        zoomTool.graphics.drawRect(0, 0, 17, 17);
        zoomTool.graphics.endFill();

        // Hide the rectangle that was the target.
        zoomWidget.visible = false;
    }

private function zoom_mouseMoveHandler(event:MouseEvent):void {
    // Update the position of the dragged rectangle.
    zoomTool.x += event.stageX - lastX;
    zoomTool.y += event.stageY - lastY;
    lastX = event.stageX;
    lastY = event.stageY;

    var bm:BitmapData = new BitmapData(16, 16);

    // Capture the bits on the screen.
    // Use the globalToLocal() method to get the coordinates of the rectangle.
    // Untrusted sub applications do not have access to the stage so you have
    // to call the globalToLocal() method on a point.
    var pt:Point = new Point(zoomTool.transform.pixelBounds.x + 2,
        zoomTool.transform.pixelBounds.y + 2);
    pt = DisplayObject(systemManager.getSandboxRoot()).globalToLocal(pt);
    bm.draw(DisplayObject(systemManager.getSandboxRoot()),
        new Matrix(1, 0, 0, 1, -pt.x, -pt.y));

    // Create a Bitmap to hold the bits.
    if (zoomed.numChildren == 0) {
        var bmp:Bitmap = new Bitmap();
        zoomed.addChild(bmp);
    } else
        bmp = zoomed.getChildAt(0) as Bitmap;

    // Set the bits.
    bmp.bitmapData = bm;

    // Zoom in on the bits.
    bmp.scaleX = bmp.scaleY = 8;
}

private function zoom_mouseUpHandler(event:Event):void {
    // Remove the listeners.
    systemManager.getSandboxRoot().removeEventListener(
        MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
    systemManager.getSandboxRoot().removeEventListener(
        MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);
    systemManager.getSandboxRoot().removeEventListener(
        SandboxMouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);

    // Replace the target rectangle.

```

```

        zoomWidget.visible = true;

        // Remove the dragged rectangle.
        PopupManager.removePopUp(zoomTool);
    }

]]>
</mx:Script>
<mx:HBox>
    <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
        <mx:Label text="Drag Rectangle"/>
        <mx:UIComponent id="zoomWidget" width="17" height="17"/>
        <mx:Canvas id="zoom"
            borderStyle="solid"
            borderThickness="2"
            width="132"
            height="132"
        >
            <mx:UIComponent id="zoomed" width="128" height="128"/>
        </mx:Canvas>
    </mx:HBox>
    <mx>List dataProvider="{data1}"/>
</mx:HBox>
</mx:Application>

```

The following example main application loads the previous example application. It sets the value of the `trustContent` property to `false` to mimic the behavior of a remotely loaded untrusted sub-application. It also links the `PopupManager` because the sub-application uses it.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern4.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" backgroundColor="#FFFFFF">
    <mx:Script>
        <![CDATA[
            /* The PopupManager must be linked in to all applications that are
               at the same security sandbox root. Because the sub application
               in this example uses the PopupManager, the main application must also
               link it it. */
            import mx.managers.PopupManager;
        ]]>
    </mx:Script>
    <mx:Text text="Portal (untrusted versioning application)"/>
    <mx:SWFLoader id="swf1"
        compatibleLoad="true"
        trustContent="false"
        source="ZoomerPattern4.swf"
    />
</mx:Application>

```

Accessing flashVars variables in sandboxed applications

You can access application parameters that were passed into the sub-application as `flashVars` variables. To do this, you access the Array of parameters on the application object by using the `getSandboxRoot()` method.

The following example gets a reference to the root application and accesses the parameters:

```
var app:DisplayObject = DisplayObject(
    SystemManager.getSandboxRoot()["application"]);
var parameters:Object = app["parameters"];
```

Developing multi-versioned applications

In most applications, even very large applications, the entire base of source code is compiled before release. This recompilation ensures that all pieces of the applications are using the same APIs. This model works well for desktop software and software whose releases are tightly controlled. But it does not always work well in the Internet model of application development, where applications are continually updated.

Internet applications are often developed and released gradually, with new pieces of functionality being added to the application all the time. To ease application development of this kind, Flex lets you create multi-versioned applications. A single application can load multiple sub-applications even if those sub-applications are compiled with different versions of the Flex framework. Using multi-versioned applications is common for sandboxed applications where more than one group of developers works on different pieces of functionality. It is also common on very large applications that are continually improved on and released.

Using RPC and DataServices-related functionality in your multi-versioned applications has some limitations. These limitations depend on the type of remote data access you use: RPC classes with a proxy server, RPC classes without a proxy server, and DataServices and RemoteObject functionality. In many of these cases, you should develop sandboxed applications rather than multi-versioned applications. For more information, see “Using RPC and DataServices classes with multi-versioned applications” on page 40.

Flex supports multi-versioned applications that were compiled with version 3.2 of the Flex framework and later. For example, your main application can be a Flex 4 application, and the loaded applications can be compiled with either version 3.2 or version 4 of the framework. If you try to load an application that was compiled by a previous version of the framework, you will get run time errors if the two applications use a shared resource whose API changed between versions of the Flex framework.

In general, a main application that is compiled for Flash Player 10 can load sub-applications that were compiled for Flash Player 10 or Flash Player 9. However, main applications that were compiled for Flash Player 9 cannot load sub-applications that were compiled for Flash Player 10.

Applications that are compiled with a different version of the Flex framework can be loaded either locally (from the same domain or subdomain, in which case they are trusted) or cross-domain (from a different domain, in which case they are untrusted by default). A trusted multi-versioned application will have nearly the same level of interoperability with the main application as a single-versioned trusted application. Styles and ResourceBundles are not shared, but otherwise they have the same level of interoperability. An untrusted multi-versioned application has slightly more interoperability than a single-versioned untrusted application has when loaded. With untrusted multi-versioned applications, focus management and mouse events work with the main application.

When you load a multi-versioned application, you set the `loadForCompatibility` property of the `SWFLoader` control to `true`. Setting this property instructs the loader to load the application into a sibling application domain. Applications that are in sibling application domains each maintain their own class definitions. They communicate through event passing, if necessary. For example, if a sub-application must do something that only the top-level application’s manager can do, then it communicates with the main application by dispatching an event.

The following sections describe how to develop multi-versioned applications. They assume that all applications are trusted (loaded into the same security domain), and loaded into sibling application domains of the main application. For information about building applications that are in separate security domains, see “Developing sandboxed applications” on page 31.

Using RPC and DataServices classes with multi-versioned applications

RPC and DataServices classes are a special case when used in multi-versioned sub-applications. All applications in the same security domain must share the same definitions of the RPC and DataServices classes for messaging to work. In addition, they might also need to share the same definitions for custom value object classes. This can be a problem because the only way to ensure that a main application and a sub-application use the same set of definitions is to load the sub-application into a child application domain of the main application, which would prevent multi-versioned applications from working. As a result, to use RPC classes in multi-versioned applications, you can do one of the following:

Load the applications as sandboxed applications If you do this, you must call the `Security.allowDomain()` method from the main application on the sub-application's domain in the SWFLoader control's `complete` event handler. You must also call the `Security.allowDomain()` method from the sub-application on the main application's domain. This provides the same level of application interoperability as multi-versioned applications. In the sub-application, you must call the `allowDomain()` method early on in the initialization cycle. Use the application's `preinitialize` event.

Use a bootstrap loader You define the messaging classes in a bootstrap loader. You then load the main application into the bootstrap loader. The main application and its sub-applications can then share those class definitions.

When you use RPC classes (such as `WebService` and `HTTPService`) without a proxy server, you do not need to load the sub-application as a sandboxed application or externalize the classes in a bootstrap class loader. Applications that use these classes without a proxy server should work as well as any multi-versioned application. Be sure to that either the URLs for the target services are on the same server as the main application, or that the target server has a `crossdomain.xml` file on it that allows access.

For more information, see “Developing sandboxed applications” on page 31 and “Bootstrap loading” on page 45.

Pop-up controls in multi-versioned applications

When a sub-application launches a pop-up control, it floats over the entire application, and can be dragged anywhere on the stage. Pop-up controls are created in the sub-application's application domain and passed to the main application's `PopUpManager`. Pop-up controls cannot be strongly-typed as `IUIComponent`, so they are marshaled from the sub-application to the main application for display.

Pop-up windows behave as follows in a multi-versioned sub-application (these behaviors are the same for single-versioned sub-applications):

- Launching a modal dialog box dims the entire application, not just the sub-application that launched it
- Centering a pop-up centers it over the entire application, not just the sub-application
- Dragging pop-up controls works over the entire application, not just the sub-application
- Focus shifts to the pop-up when you first launch a pop-up from the sub-application

The `PopUpManager` must be linked in to all applications that are sandbox roots. When using the `PopUpManager` in a child application, there must also be an instance of the `PopUpManager` in the main application. To link the `PopUpManager` to a main application that doesn't use it, you can add the following code:

```
import mx.managers.PopUpManager;  
private var popupManager1:PopUpManager;
```

If you do not link a `PopUpManager` into an application, and a sub-application in its sandbox requests that it display a modal dialog box, you might get the following error:

```
No class registered for interface 'mx.managers::IPopUpManager'.
```

List controls require some additional code when used in sub-applications. If you have a List control in a sub-application's pop-up and no List control in your main application, you must also link the List class into your main application. In addition, the List control must have a data provider. In this case, you can set it to an empty array, as the following example shows:

```
<mx:List visible="false" includeInLayout="false" dataProvider="[]" />
```

Embedded fonts in multi-versioned applications

Applications that are in the same application domain are able to use the same embedded fonts by specifying the font name. However, if the sub-application is loaded into a different application domain (as is the case with multi-versioned applications), then the sub-application must embed the font to use it.

Styles and style modules in multi-versioned applications

Using styles and style modules in multi-versioned sub-applications is the same as using them in sandboxed sub-applications. For more information on using styles and style modules in sub-applications, see “Styles and style modules in sandboxed applications” on page 33.

Focus in multi-versioned applications

Using the FocusManager in multi-versioned sub-applications is the same as using the FocusManager in sandboxed sub-applications. (See “Focus in sandboxed applications” on page 33.)

Cursors in multi-versioned applications

The CursorManager class in the main application and sub-application communicate much like the PopupManager class. If the sub-application changes the cursor to a custom cursor, this custom cursor continues to appear when the cursor is moved outside the sub-application and over the main application. Calling a method or setting a property on the CursorManager in the sub-application bubbles up to the main application, and vice versa.

If a busy cursor is visible while the mouse is over the sub-application, and you move the cursor to the main application, the cursor remains as a busy cursor.

Localizing multi-versioned applications

Using resource bundles in multi-versioned sub-applications is the same as using them in sandboxed sub-applications. For more information on using resource bundles in sub-applications, see “Localizing sandboxed applications” on page 34.

ToolTip object in multi-versioned applications

ToolTip objects are created in the sub-application's application domain and are passed to the main application's ToolTipManager. This is an example of marshaling a class across application boundaries.

This behavior applies to error tips and data tips on List objects in multi-versioned applications as well.

Layouts in multi-versioned applications

The main application dictates the area of the screen that is available to the sub-application for screen layout.

LayoutManagers in different applications run independent of one another. The applications are not clipped.

Deep linking in multi-versioned applications

You cannot access the main application's `BrowserManager` directly from a sub-application that is in a different application domain. The sub-application's `BrowserManager` is disabled. This is the same for sandboxed and multi-versioned sub-applications. For more information, see "Deep linking in sandboxed applications" on page 35.

Dragging and dropping in multi-versioned applications

When the sub-application is in the same security domain as the main application, but in a different application domain, you can drag list items from a sub-application to a list in the main application, and vice versa. For this to work across applications, the class that defines the drop target must be public.

The `DragProxy` maintains the item's appearance during the entire event, regardless of which application the mouse is over. Drag events are triggered for all source and destination controls, as if they are in the same application.

Behind the scenes, the `DragManager` generates the `DragProxy` in the sub-application's application domain then passes that `DragProxy` to the main application's `DragManager` during the drag operation.

Strongly-typed objects cannot be passed from the main application to the sub-application or vice versa. As a result, the `DragManager` accepts a `DragProxy` that is not strongly typed.

All properties and methods defined by the `IDragManager` interface are handled through delegation to the main application's `DragManager`.

If you drag a generic type object, the object will be of generic type `Object` in Flash Player's application domain. Flex marshals the properties so that the drag and drop operation works without having to write any custom code.

If you use a custom class as part of the drop data, that class cannot be shared as strongly-typed. In that case, override the drop events and add logic to marshal the data from the drag source to the drop target.

Listening for mouse events in multi-versioned applications

Mouse interaction between a main application and a sub-application can be confusing, especially when those events occur in different application domains. For example, when you click and drag an object in a sub-application, and then release the button over the main application, typical Flex mouse events such as `MOUSE_UP`, `MOUSE_DOWN_OUTSIDE`, or `MOUSE_LEAVE` are triggered but cannot be listened to directly in the sub-application.

To listen for mouse events across application domains, listen to all mouse activity in the security domain. To listen, get a reference to the sandbox root and register your event listeners with that `SystemManager`.

When loaded into a separate application domain, the `topLevelSystemManager` property refers to the sub-application's `SystemManager` because the main application's `SystemManager` is in another application domain. As a result, you do not use the `topLevelSystemManager` property to get a reference to the main application's `SystemManager`. Instead, you use the `systemManager.getSandboxRoot()` method to get a reference to the top-level `SystemManager` in the security domain. From this `SystemManager`, you can access all mouse activity in the current security domain.

The following example uses calls to the `getSandboxRoot()` method to get references to the top-level `SystemManager` in the security domain.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/ZoomerPattern3.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="setup()"
    height="250"
>
    <mx:Script>
    <![CDATA[
        import mx.core.UIComponent;
        import mx.managers.PopUpManager;

        [Bindable]
        public var data:Array = ["Ice Cream", "Fudge", "Whipped Cream", "Nuts"];

        public var zoomTool:UIComponent;

        public function setup():void {
            // Draw the zoom rectangle.
            zoomWidget.graphics.lineStyle(1);
            zoomWidget.graphics.beginFill(0, 0);
            zoomWidget.graphics.drawRect(0, 0, 17, 17);
            zoomWidget.graphics.endFill();

            // Listen for mouse down events.
            zoomWidget.addEventListener(MouseEvent.CLICK, zoom_mouseDownHandler);
        }

        private var lastX:int;
        private var lastY:int;

        private function zoom_mouseDownHandler(event:MouseEvent):void {
            // When the mouse is down, listen for the move and up events.
            // The getSandboxRoot() method lets you listen to all mouse activity in your
            // SecurityDomain.
            systemManager.getSandboxRoot().addEventListener(
                MouseEvent.CLICK, zoom_mouseClickHandler, true);
            systemManager.getSandboxRoot().addEventListener(
                MouseEvent.CLICK, zoom_mouseUpHandler, true);

            // Update last position of the mouse.
            lastX = event.stageX;
            lastY = event.stageY;

            // Create and pop up the zoomTool. This is the rectangle that is dragged around.
            // It must be a popup so that it can float over other content.
            zoomTool = new UIComponent();
            PopUpManager.addPopUp(zoomTool, this);

            var pt:Point = new Point(zoomWidget.transform.pixelBounds.x,
                zoomWidget.transform.pixelBounds.y);
            pt = zoomTool.parent.globalToLocal(pt);
            zoomTool.x = pt.x;
            zoomTool.y = pt.y;
            zoomTool.graphics.lineStyle(1);
            zoomTool.graphics.beginFill(0, 0);
            zoomTool.graphics.drawRect(0, 0, 17, 17);
        }
    ]]>
    </mx:Script>
</mx:Application>

```

```
zoomTool.graphics.endFill();

// Hide the rectangle that was the target.
zoomWidget.visible = false;
}

private function zoom_mouseMoveHandler(event:MouseEvent):void {
    // Update the position of the dragged rectangle.
    zoomTool.x += event.stageX - lastX;
    zoomTool.y += event.stageY - lastY;
    lastX = event.stageX;
    lastY = event.stageY;

    var bm:BitmapData = new BitmapData(16, 16);

    // Capture the bits on the screen.
    // You must use the getSandboxRoot() method here, too, because it gets
    // the parent of all of the pixels you are allowed to access.
    bm.draw(DisplayObject(systemManager.getSandboxRoot()),
        new Matrix(1, 0, 0, 1, -zoomTool.transform.pixelBounds.x - 2,
            -zoomTool.transform.pixelBounds.y - 2));

    // Create a Bitmap to hold the bits.
    if (zoomed.numChildren == 0) {
        var bmp:Bitmap = new Bitmap();
        zoomed.addChild(bmp);
    } else
        bmp = zoomed.getChildAt(0) as Bitmap;

    // Set the bits.
    bmp.bitmapData = bm;

    // Zoom in on the bits.
    bmp.scaleX = bmp.scaleY = 8;
}

private function zoom_mouseUpHandler(event:Event):void {
    // Remove the listeners.
    systemManager.getSandboxRoot().removeEventListener(
        MouseEvent.MOUSE_MOVE, zoom_mouseMoveHandler, true);
    systemManager.getSandboxRoot().removeEventListener(
        MouseEvent.MOUSE_UP, zoom_mouseUpHandler, true);

    // Replace the target rectangle.
    zoomWidget.visible = true;

    // Remove the dragged rectangle.
    PopupManager.removePopup(zoomTool);
}
```

```

    }

    ]]>
</mx:Script>
<mx:HBox>
    <mx:HBox backgroundColor="0x00eeee" height="140" paddingTop="4" paddingRight="4">
        <mx:Label text="Drag Rectangle"/>
        <mx:UIComponent id="zoomWidget" width="17" height="17"/>
        <mx:Canvas id="zoom"
            borderStyle="solid"
            borderThickness="2"
            width="132"
            height="132"
        >
            <mx:UIComponent id="zoomed" width="128" height="128"/>
        </mx:Canvas>
    </mx:HBox>
    <mx>List dataProvider="{data1}"/>
</mx:HBox>
</mx:Application>

```

The following example main application loads the previous example application. It sets the `loadForCompatibility` property to `true` so that the sub-application loads a multi-versioned application.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- apploading/MainZoomerPattern3.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" backgroundColor="#FFFFFF">
    <mx:Text text="Cross-Versioning (trusted versioning application):"/>
    <mx:SWFLoader id="swf1" compatibleLoad="true" source="ZoomerPattern3.swf"/>
</mx:Application>

```

Bootstrap loading

A bootstrap loader is a lightweight application that defines classes that you want to share among applications.

The bootstrap loader gets loaded first instead of the actual main application, and then loads the main application into a child application domain of itself. The main application then loads a sub-application into a sibling application domain. The two applications are both loaded into child application domains of the bootstrap loader's application domain. The result is that they share any classes that are defined in the bootstrap, but use their own class definitions for everything else.

You can typically only have one bootstrap loader in a security domain. Bootstrapped classes are not shared across security domains.

Changes to any class in a bootstrap loader can break other sub-applications. As a result, ensure that the classes you add to a bootstrap loader are not likely to change.

When you use RPC classes with a proxy server (such as Blaze DS or LCDS), you must externalize the RPC classes in a bootstrap loader for multi-versioned applications. The bootstrap loader should contain only the RPC classes. You typically do not add your custom value object classes to the bootstrap loader to get or put data on the server. You can add those classes if you want to reduce marshalling, but in that case they should be simple classes that do not link in Flex framework classes such as `Array` or `ArrayCollection`.

When you use DataServices or RemoteObject functionality, you typically use a bootstrap loader for loading multi-versioned applications. The bootstrap loader should contain the necessary RPC and DataServices-related classes. It must also contain any value object classes that are used to send data to or from the server. If you add custom value object classes to your bootstrap loader, they should be simple classes that do not link in Flex framework classes such as Array or ArrayCollection. Adding complex value object classes to the bootstrap loader is not supported.

The following is an example of the bootstrap loader for RPC classes. This kind of bootstrap loader is commonly used when a sub-application uses classes such as HTTPService and WebService with a proxy server.

Take note of the way that the MainApp.swf is loaded. The bootstrap loader uses a Loader class, and does not define the LoaderContext, which results in default behavior for the Loader. The defaults are to load a sub-application into a child application domain, and to assume the defaults for the security domain.

Notice, too, that for each import, a definition of the class immediately follows the class's import statement exists. The result is that the class definitions are linked into the bootstrap loader. The loaded application, and all subsequently loaded applications, then share those definitions.

```
package {

import flash.display.Loader;
import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
import flash.events.Event;
import flash.net.URLRequest;
import flash.system.ApplicationDomain;

/**
 * Classes used by the networking protocols go here. These are the classes
 * whose definitions are added to the bootstrap loader and then shared
 * by the main application and all sub applications.
 */
import mx.messaging.config.ConfigMap; ConfigMap;
import mx.messaging.messages.AcknowledgeMessage; AcknowledgeMessage;
import mx.messaging.messages.AcknowledgeMessageExt; AcknowledgeMessageExt;
import mx.messaging.messages.AsyncMessage; AsyncMessage;
import mx.messaging.messages.AsyncMessageExt; AsyncMessageExt;
import mx.messaging.messages.CommandMessage; CommandMessage;
import mx.messaging.messages.CommandMessageExt; CommandMessageExt;
import mx.messaging.messages.ErrorMessage; ErrorMessage;
import mx.messaging.messages.HTTPRequestMessage; HTTPRequestMessage;
import mx.messaging.messages.MessagePerformanceInfo; MessagePerformanceInfo;
import mx.messaging.messages.RemotingMessage; RemotingMessage;
import mx.messaging.messages.SOAPMessage; SOAPMessage;
import mx.messaging.channels.HTTPChannel; HTTPChannel;
import mx.core.mx_internal;

[SWF(width="600", height="700")]
public class RPCBootstrapLoader extends Sprite {
    /**
     * The URL of the application SWF that this bootstrap loader loads.
     */
    private static const applicationURL:String = "MainApp.swf";

    /**
     * Constructor.
     */
}
```

```

public function RPCBootstrapLoader() {
    super();

    if (ApplicationDomain.currentDomain.hasDefinition("mx.core::UIComponent"))
        throw new Error("UIComponent should not be in the bootstrap loader.");

    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.align = StageAlign.TOP_LEFT;

    if (!stage)
        isStageRoot = false;

    root.loaderInfo.addEventListener(Event.INIT, initHandler);
}

/**
 * The Loader that loads the main application's SWF file.
 */
private var loader:Loader;

/**
 * Whether the bootstrap loader is at the stage root or not.
 * It is only the stage root if it was the root
 * of the first SWF file that was loaded by Flash Player.
 * Otherwise, it could be a top-level application but not stage root
 * if it was loaded by some other non-Flex shell or is sandboxed.
 */
private var isStageRoot:Boolean = true;

/**
 * Called when the bootstrap loader's SWF file has been loaded.
 * Starts loading the application SWF specified by the applicationURL property.
 */
private function initHandler(event:Event):void {
    loader = new Loader();
    addChild(loader);
    loader.load(new URLRequest(applicationURL));
    loader.addEventListener("mx.managers.SystemManager.isBootstrapRoot",
        bootstrapRootHandler);
    loader.addEventListener("mx.managers.SystemManager.isStageRoot",
        stageRootHandler);

    stage.addEventListener(Event.RESIZE, resizeHandler);
}

```

```
private function bootstrapRootHandler(event:Event):void {  
    // Cancel event to indicate that the message was heard.  
    event.preventDefault();  
}  
  
private function stageRootHandler(event:Event):void {  
    // Cancel event to indicate that the message was heard.  
    if (!isStageRoot)  
        event.preventDefault();  
}  
  
private function resizeHandler(event:Event):void {  
    loader.width = stage.width;  
    loader.height = stage.height;  
    Object(loader.content).setActualSize(stage.width, stage.height);  
}  
}  
}
```

The following image shows where each domain gets its class definitions for a bootstrap loader that defines the `mx.messaging.*` classes.

