

You can create modules that you dynamically load in your Flex applications.

Contents

Modular applications overview	1095
Creating modules	1098
Compiling modules	1102
Loading and unloading modules	1110
Using ModuleLoader events	1119
Passing data	1128

Modular applications overview

This section describes modules and how they are used by modular applications.

About modules

Modules are SWF files that can be loaded and unloaded by an application. They cannot be run independently of an application, but any number of applications can share the modules.

Modules let you split your application into several pieces, or modules. The main application, or shell, can dynamically load other modules that it requires, when it needs them. It does not have to load all modules when it starts, nor does it have to load any modules if the user does not interact with them. When the application no longer needs a module, it can unload the module to free up memory and resources.

Modular applications have the following benefits:

- Smaller initial download size of the SWF file.
- Shorter load time due to smaller SWF file size.

- Better encapsulation of related aspects of an application. For example, a “reporting” feature can be separated into a module that you can then work on independently.

Benefits of modules

A module is a special type of dynamically-loadable SWF that contains an `IFlexModuleFactory` class factory. This allows an application to load code at run time and create class instances without requiring that the class implementations be linked into the main application.

Modules are similar to Runtime Shared Libraries (RSLs) in that they separate code from an application into separately loaded SWF files. Modules are much more flexible than RSLs because modules can be loaded and unloaded at run time and compiled without the application.

Two common scenarios in which using modules is beneficial are a large application with different user paths and a portal application.

An example of the first common scenario is an enormous insurance application that includes thousands of screens, for life insurance, car insurance, health insurance, dental insurance, travel insurance, and veterinary pet insurance.

Using a traditional approach to rich Internet application (RIA) design, you might build a monolithic application with a hierarchical tree of MXML classes. Memory use and start-up time for the application would be significant, and the SWF file size would grow with each new set of functionality.

When using this application, however, any user accesses only a subset of the screens. By refactoring the screens into small groups of modules that are loaded on demand, you can improve the perceived performance of the main application and reduce the memory use. Also, when the application is separated into modules, developers’ productivity may increase due to better encapsulation of design. When rebuilding the application, the developers also have to recompile only the single module instead of the entire application.

An example of the second common scenario is a system with a main portal application, written in ActionScript 3, that provides services for numerous portlets. Portlets are configured based on data that is downloaded on a per-user basis. Using the traditional approach, you might build an application that compiles in all known portlets. This is inefficient, both for deployment and development.

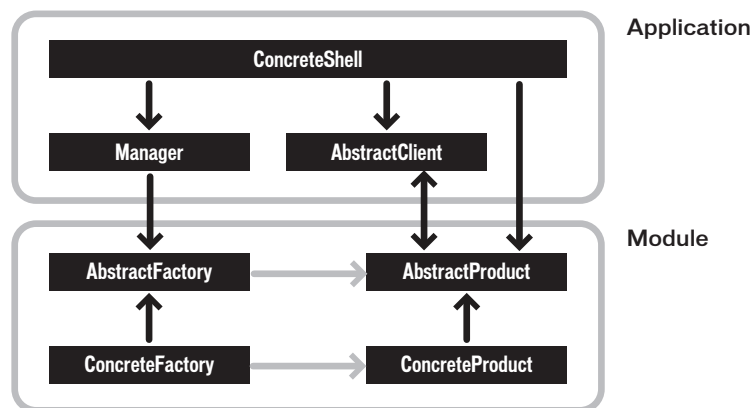
By using modules, you can establish an interface that contains portal services, and a generic portlet interface. You can use XML data to determine which modules to load for a given session. When the module is loaded, you obtain a handle to a class factory inside the module, and from that you create an instance of a class that implements the portlet interface. In this scenario, full recompilation is necessary only if the interfaces change.

Module API details

Modules implement a class factory with a standard interface. The product of that class factory implements an interface known to the shell, or the shell implements an interface known to the modules.

By using shared interface definitions, these shared interfaces reduce hard dependencies between the shell and the module. This provides type-safe communication and enforces an abstraction layer without adding significantly to the SWF file size.

The following image shows the relationship between the shell and the module's interfaces:



The ModuleManager manages the set of loaded modules, which are treated as a map of singletons indexed by the module URL. Loading a module triggers a series of events that let clients monitor the status of the module. Modules are only ever loaded once, but subsequent reloads also dispatch events so that client code can be simplified and rely on using the `READY` event to know that the module's class factory is available for use.

The ModuleLoader class is a thin layer on top of the ModuleManager API that is intended to act similarly to the `mx.controls.SWFLoader` class for modules that only define a single visual UIComponent. The ModuleLoader class is the easiest class to use when implementing a module-based architecture, but the ModuleManager provides greater control over the modules.

Creating modular applications

To create a modular application, you create separate classes for each module, plus an application that loads the modules.

To create a modular application:

1. Create any number of modules. An MXML-based module file's root tag is `<mx:Module>`. ActionScript-based modules extend either the `Module` or `ModuleBase` class.
2. Compile each module as if it were an application. You can do this by using the `mxmhc` command-line compiler or the compiler built into Adobe Flex Builder.
3. Create an Application class. This is typically an MXML file whose root tag is `<mx:Application>`, but it can also be an ActionScript-only application.
4. In the Application file, use an `<mx:ModuleLoader>` tag to load each of the modules. You can also load modules by using methods of the `mx.modules.ModuleLoader` and `mx.modules.ModuleManager` classes.

The following sections describes these steps in detail.

Creating modules

Modules are classes just like application files. You can create them in either ActionScript or by extending a Flex class by using MXML tags. This section describes how to create modules in MXML and in ActionScript.

Creating MXML-based modules

To create a module in MXML, you extend the `mx.modules.Module` class by creating a file whose root tag is `<mx:Module>`. In that tag, ensure that you add any namespaces that are used in that module. You must also include an XML type declaration tag at the beginning of the file, such as the following:

```
<?xml version="1.0"?>
```

The following example is a module that includes a Chart control:

```
<?xml version="1.0"?>
<!-- modules/ColumnChartModule.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" >
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;
    [Bindable]
    public var expenses:ArrayCollection = new ArrayCollection([
      {Month:"Jan", Profit:2000, Expenses:1500},
      {Month:"Feb", Profit:1000, Expenses:200},
      {Month:"Mar", Profit:1500, Expenses:500}
    ]);
  ]]></mx:Script>
  <mx:ColumnChart id="myChart" dataProvider="{expenses}">
    <mx:horizontalAxis>
      <mx:CategoryAxis
        dataProvider="{expenses}"
        categoryField="Month"
      />
    </mx:horizontalAxis>
    <mx:series>
      <mx:ColumnSeries
        xField="Month"
        yField="Profit"
        displayName="Profit"
      />
      <mx:ColumnSeries
        xField="Month"
        yField="Expenses"
        displayName="Expenses"
      />
    </mx:series>
  </mx:ColumnChart>
  <mx:Legend dataProvider="{myChart}"/>
</mx:Module>
```

After you create a module, you compile it as if it were an application. For more information on compiling modules, see “Compiling modules” on page 1102.

After you compile a module, you can load it into an application or another module. Typically, you use one of the following techniques to load MXML-based modules:

- **ModuleLoader** — The `ModuleLoader` class provides the highest-level API for handling modules. For more information, see “Using the `ModuleLoader` class to load modules” on page 1111.
- **ModuleManager** — The `ModuleManager` class provides a lower-level API for handling modules than the `ModuleLoader` class does. For more information, see “Using the `ModuleManager` class to load modules” on page 1114.

Creating modules in Flex Builder

Flex Builder does not include explicit support for the module type. As a result, you should treat modules as applications in Flex Builder. The following steps describe how to add a new application to your MXML project in Flex Builder and then convert that application to a module. However, you can also create a new project for each module. For more information, see “Compiling modules with Flex Builder” on page 1103.

To add a module to your project in Flex Builder:

1. In Flex Builder, open your MXML application’s project file.
2. Select File > New > MXML Application.
3. Enter a filename for the new module. For example, MyModule.
4. Click the Finish button. Flex Builder adds a new MXML application file in your project, in the same directory as your application.

The following example shows the default contents of this new application:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

</mx:Application>
```

5. To convert this new application to a module, replace the `<mx:Application>` tag with the `<mx:Module>` tag so that the resulting code looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">

</mx:Module>
```

By default, whenever you add an MXML application file to your Flex project, it is runnable and is added to the list of project application files. All files defined as application files must reside in your project’s source folder. Because modules must be runnable, they must be in your project’s source folder.

To determine if a file is runnable, you can examine the file’s icon in the Navigator. A file with a green arrow is runnable. A file without a green arrow is not runnable.

You can also add a module to your Flex Builder project by selecting File > New > MXML Component. You then select Module from the Based On drop-down list. While this creates a new file with a root tag of `<mx:Module>` rather than `<mx:Application>`, it does not make the new file runnable. To make a new MXML component runnable, you must edit the list of runnable files.

To edit the list of runnable application files:

1. Right-click the project file and select properties.
2. Select the Flex Applications option.

The list of Flex applications appears in the dialog box. This list includes all MXML files that were added as applications, regardless of their root tags.

You can add and remove modules from this dialog box by using the Add and Remove buttons.

Creating ActionScript-based modules

To create a module in ActionScript, you can either create a file that extends the `mx.modules.Module` class or the `mx.modules.ModuleBase` class.

Extending the `Module` class is the same as using the `<mx:Module>` tag in an MXML file. You extend this class if your module interacts with the framework; this typically means that it adds objects to the display list or otherwise interacts with visible objects.

To see an example of an ActionScript class that extends the `Module` class, create an MXML file with the root tag of `<mx:Module>`. When you compile this file, set the value of the `keep-generated-actionscript compiler` property to `true`. The Flex compiler stores the generated ActionScript class in a directory called `generated`. You will notice that this generated class contains code that you probably will not understand. As a result, you should not write ActionScript-based modules that extend the `Module` class; instead, you should use MXML to write such modules.

If your module does not include any framework code, you can create a class that extends `ModuleBase`. If you use the `ModuleBase` class, your module will typically be smaller than if you use a module based on the `Module` class because it does not have any framework class dependencies.

The following example creates a simple module that does not contain any framework code and therefore extends the `ModuleBase` class:

```
// modules/asmodules/SimpleModule.as
package {
    import mx.modules.ModuleBase;

    public class SimpleModule extends ModuleBase {
        public function SimpleModule() {
            trace("SimpleModule created");
        }

        public function computeAnswer(a:Number, b:Number):Number {
            return a + b;
        }
    }
}
```

To call the `computeAnswer()` method on the ActionScript module, you can use one of the techniques shown in “Accessing modules from the parent application” on page 1129.

Compiling modules

You compile modules the same way you compile Flex applications. On the command line, you use the `mxmcl` command-line compiler. In Flex Builder, you create modules as applications and compile them by either building the project or running the application.

The result is a SWF file that you load into your application as a module. You cannot run the module-based SWF file as a stand-alone Flash application or load it into a browser window. It must be loaded by an application as a module. When you run it in Flex Builder to compile it, you should close the Player or browser Window and ignore any errors. Modules should not be requested by the Player or through a browser directly.

For more information on compiling modules on the command line, see “Compiling modules with the command-line compiler” on page 1103.

For information on compiling modules in Flex Builder, see “Compiling modules with Flex Builder” on page 1103.

After you compile your module, you try to remove redundancies between the module and the application that uses it. To do this, you create a link report for the application, and then externalize any assets in the module that appear in that report. For more information, see “Reducing module size” on page 1108.

Compiling modules with the command-line compiler

You compile the module as you would compile any Flex application using the `mxm1c` command-line compiler or the Flex Builder compiler. The following command is the simplest `mxm1c` command:

```
mxm1c MyModule.mxml
```

Once you compile a module, you should try to reduce its size by removing dependencies that overlap with the application that uses the module. For more information, see “Reducing module size” on page 1108.

Compiling modules with Flex Builder

Flex Builder does not contain explicit support for creating and compiling modules. As a result, setting up a project that uses modules is not entirely intuitive. When setting up projects that use modules, you can treat the modules as MXML applications within the application’s project or as separate projects. This section describes how to do both of these.

Once you compile a module, you should try to reduce its size by removing dependencies that overlap with the application that uses the module. For more information, see “Reducing module size” on page 1108.

Overview

There are two approaches to using modules in Flex Builder projects:

- **Single project** — In this approach, you create a single project for your application, and then add modules to the application list. This approach provides less customization and is more limiting than creating modules in separate projects. For more information, see “Using modules in a single project” on page 1103.
- **Multiple projects** — In this approach, you create a separate project for each module or a single project for all the modules. This approach can be appropriate for when you only have a few modules or need to customize the compilation of your modules. For more information, see “Using multiple projects” on page 1104.

Using modules in a single project

You can create modules in the same project as your application and add them to the application list.

Using a single project for all application and module files has the following benefits:

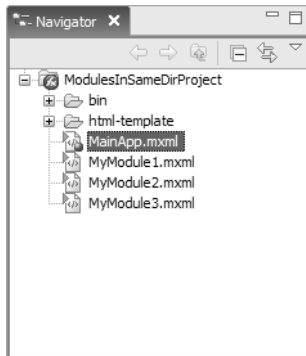
- A single project takes less memory than multiple projects.

- The modules are automatically recompiled when the application is compiled.
- You can compile modules individually.

This approach has the following drawbacks:

- The application and the modules must share the same compiler settings (such as the same library path).
- The application and the module files must be in the same source directory.
- You cannot use the `load-externs` compiler option to remove overlapping dependencies without compiling the modules separately using the command-line compiler.

You can compile the modules at the same time as your application, as long as the modules are in the same root directory as the application and they are marked as runnable. The following example shows the structure of a project that has several modules in the same root directory as the application:



To add a new module to your project, see “Creating MXML-based modules” on page 1098.

When you run your application, Flex Builder compiles the modules’ SWF files for you. The SWF files are then loaded into the application at run time. The module SWF files and main application SWF file are in the same directory.

Using multiple projects

You can create a separate Flex or ActionScript project for each module. This gives you greater control over how modules are compiled because each project can have different compiler options than the application or other modules. You can also choose to compile the module’s project(s) without compiling the application.

However, this approach requires that you manually compile each module before compiling the application unless you compile all open projects in Flex Builder at one time.

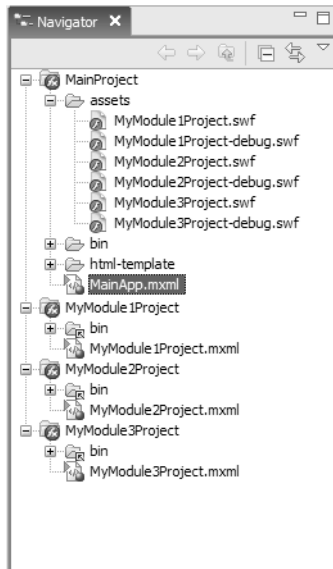
Using one project for each module has the following benefits:

- Module projects can be located anywhere in the workspace.
- Module projects can have their own compiler settings, such as a custom library path.
- Module projects can use the `load-externs` compiler option to remove overlapping dependencies.

Using one project for each module has the following drawbacks:

- Having many projects uses more memory.
- Having many projects in a single workspace can make the workspace crowded.
- By default, when you compile the application, all module projects are not compiled even if they have changed.

The following example has a main project and three module projects. Each module project compiles and outputs its SWF files to the main project's assets directory:



A related approach is to use a single project for all modules, while keeping the application in its own separate project. This has some of the drawbacks of using a single project for both the application and the modules, but has many of the same benefits as using a separate project for each of the modules.

Using one project for all modules has the following benefits:

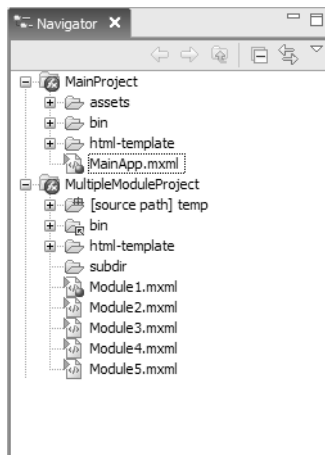
- The module project can be located anywhere in the workspace.
- You can compile just the modules or just the application, without having to recompile both at the same time.

- The module project can use the `load-externs` compiler option to remove overlapping dependencies.

Using one module project for all modules has the following drawbacks:

- All the modules in the module project must use the same compiler settings, such as the library path.
- By default, when you compile the application, the module project is not compiled even if the module project has changed.

The following example has a main project and a module projects. The module project compiles and outputs its SWF files to the main project's assets directory:



Creating projects

When creating a separate project for modules, you change the module project's output folder to a directory that is used by the application. You also suppress the generation of wrapper files.

To create a separate project for modules in Flex Builder:

1. Create a main project.
2. Create a new project for your module or modules.
3. Right click the module's project and select Properties. The Properties dialog box displays.
4. Select the Flex Build Path option.
5. Change the Output Folder to point to the Main Project's modules directory. For example, change it to the following:

```
${DOCUMENTS}\MainProject\assets
```

This redirects the output of your module's compilation to your application project's (called `MainProject`) assets directory. In your main application, you can point the `ModuleLoader`'s `url` property to the SWF files in the assets directory.

6. Click the Apply button to save your changes.
7. In the Properties dialog box, select the Flex Compiler option.
8. Deselect the Generate HTML Wrapper File option. This prevents the module's project from generating the HTML wrapper files. You typically only use these files for the application. For modules, they are not necessary because clients should never invoke modules directly.
9. Click OK to apply the changes.

Compiling projects

Compiling multiple projects in Flex Builder is a common operation. First, you choose the order in which you want the projects to compile and then you compile all projects at the same time.

To compile all projects at the same time in Flex Builder:

1. Select `Project > Build All` from the main menu.

Flex builds all projects in the workspace. The application files are added to each project's output folder. You are then prompted to save files if you haven't already chosen to save files automatically before a build begins.

If you want to change the build order, you use the Build Order dialog box. This is not always necessary. Projects that use modules only need to be compiled by the time the main project application runs, not as it compiles. In most cases, the default build order is adequate.

However, if you want to eliminate overlapping dependencies, you might need to change the build order so that the main application compiles first. At that time, you use the `link-report` compiler option to generate the linker report. When you compile the modules, you then use the `load-externs` compiler option to use the linker report that was just generated by the shell application. For more information on reducing module size, see "Reducing module size" on page 1108.

To change the build order of the projects:

1. Select `Window > Preferences` from the main menu.

The Flex Builder Preferences dialog box appears.

2. Select `General > Workspace > Build Order`.

The Build Order dialog box displays.

3. Deselect the Use Default Build Order checkbox.

4. Reorder the projects in the Project Build Order list by using the Up and Down buttons. You can also remove projects that are not part of your main application or are not modules used by the application by using the Remove button. The removed project will still be built, but only after all the projects in the build order list are built.
5. Click OK.
6. Modify the build order as needed, and click OK.

If you create dependencies between separate projects in the workspace, the compiler automatically determines the order in which the projects are built, so these dependencies resolve properly. For more information about building projects in Flex Builder, see *Using Flex Builder 2*.

When you use a separate project for each module, you can compile a single module at a time. This can save time over compiling all projects at once, or over compiling a single project that contains all module and application files.

To compile a single module's project:

1. Right-click the module's MXML file in the module's project.
2. Select Run Application. The Player or browser window tries to run the module after it compiles. You can close the Player or browser window and ignore any error messages that appear at run time. Modules are not meant to be run in the Player or in a browser directly.

Reducing module size

Module size varies based on the components and classes that are used in the module. By default, a module includes all framework code that its components depend on, which can cause modules to be large by linking classes that overlap with the application's classes.

To reduce the size of the modules, you can instruct the module to externalize classes that are included by the application. This includes custom classes and framework classes. The result is that the module only includes the classes it requires, while the framework code and other dependencies are included in the application.

To externalize framework classes, you generate a linker report from the application that loads the modules. You then use this report as input to the module's `load-externs` compiler option.

To create and use a linker report with the command-line compiler:

1. Generate the linker report and compile the application:

```
mxmhc -link-report=report.xml MyApplication.mxml
```

The default output location of the linker report is the same directory as the compiler. In this case, it would be in the bin directory.

2. Compile the module and pass the linker report to the `load-externs` option:

```
mxmlc -load-externs=report.xml MyModule.mxml
```

If you are using Flex Builder to compile modules, you can use the technique described in this section if you have a separate project for your modules. In that case, for the main application, you add the `link-report` option to the Additional Compiler Arguments field of the Flex Compiler options. You then use the `load-externs` option in the same field for each module's project to load this report.

To create and use a linker report with modules in Flex Builder:

1. Select the main application's project in the Navigator.
2. Click Project > Properties.
3. Select Flex Compiler.
4. In the Additional Compiler Arguments field, specify an output location for the `link-report` compiler option, as the following example shows:

```
-locale en_US -link-report=c:/temp/externs/report.xml
```

The default output location of the `link-report` option in Flex Builder is the root directory of the Flex Builder application itself. On Windows, the default directory is `c:\Program Files\Adobe\Flex Builder 2`. As a result, if you add the following command to your compiler options, Flex Builder writes the `report.xml` file to its root directory:

```
-link-report=report.xml
```

5. Click OK to save your changes.
 6. Select the module's project in the Navigator. If you are using multiple projects for modules, perform the following steps for each project.
 7. Click Project > Properties.
 8. Select Flex Compiler.
 9. In the Additional Compiler Arguments field, load the linker report by using the `load-externs` compiler option, as the following example shows:
- ```
-locale en_US -load-externs=c:/temp/externs/report.xml
```
10. Click OK to save your changes.
  11. Ensure that the main application's project appears first in the build order, as described in "Compiling projects" on page 1107. This ensures that the linker report is generated by the shell application before it is consumed by the modules.
  12. Compile all projects "Compiling projects" on page 1107.

## Recompiling modules

If you change a module, you do not have to recompile the application that uses the module. This is because the application loads the module at run time and does not check against it at compile time. Similarly, if you make changes to the application, you do not have to recompile the module. Just as the application does not check against the module at compile time, the module does not check against the application until run time.

However, if you make changes that might affect the linker report or common code, you should recompile both the application and the modules.

NOTE

If you externalize the module's dependencies by using the `load-externs` option, your module might not be compatible with future versions of Adobe Flex. You might be required to recompile the module. To ensure that a future Flex application can use a module, compile that module with all the classes it requires. This also applies to applications that you load inside other applications.

## Debugging modules

To debug the application with modules, you reference the debug SWF file in the source code during development, and switch to the non-debug SWF file when you are ready to deploy. For example, when you debug, your calls to the `loadModule()` method might appear as follows:

```
loadModule('ChartModule-debug.swf')
```

When you have finished debugging and want to deploy the application, you then remove references to the debug SWF files. Your calls to the `loadModule()` method might appear as follows:

```
loadModule('ChartModule.swf')
```

The technique described in this section also applies to the SWF file names that you pass to the `ModuleLoader`'s `url` property, or any other way you refer to the URL of the module.

## Loading and unloading modules

There are several techniques you can use to load and unload modules in your Flex applications. These techniques include:

- **ModuleLoader** — The `ModuleLoader` class provides the highest-level API for handling modules. For more information, see “Using the `ModuleLoader` class to load modules” on page 1111.

- **ModuleManager** — The `ModuleManager` class provides a lower-level API for handling modules than the `ModuleLoader` class does. For more information, see “Using the `ModuleManager` class to load modules” on page 1114.

## Using the `ModuleLoader` class to load modules

You can use the `ModuleLoader` class to load module in an application or other module. The easiest way to do this in an MXML application is to use the `<mx:ModuleLoader>` tag. You set the value of the `url` property to point to the location of the module’s SWF file. The following example loads the module when the application first starts:

```
<?xml version="1.0"?>
<!-- modules/MySimplestModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:ModuleLoader url="ColumnChartModule.swf"/>
</mx:Application>
```

You can change when the module loads by setting the value of the `url` property at some other time, such as in response to an event.

Setting the target URL of a `ModuleLoader` triggers a call to the `loadModule()` method. This occurs when you first create a `ModuleLoader` with the `url` property set. It also occurs if you change the value of that property.

If you set the value of the `url` property to an empty string (“”), the `ModuleLoader` unloads the current module.

You can have multiple `ModuleLoader` instances in a single application. The following example loads the modules when the user navigates to the appropriate tabs in the `TabNavigator` container:

```
<?xml version="1.0"?>
<!-- modules/URLModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Panel
 title="Module Example"
 height="90%"
 width="90%"
 paddingTop="10"
 paddingLeft="10"
 paddingRight="10"
 paddingBottom="10"
 >

 <mx:Label width="100%" color="blue"
 text="Select the tabs to change the panel."/>

 <mx:TabNavigator id="tn"
 width="100%"
 height="100%"
 creationPolicy="auto"
 >
 <mx:VBox id="vb1" label="Column Chart Module">
 <mx:Label id="l1" text="ColumnChartModule.swf"/>
 <mx:ModuleLoader url="ColumnChartModule.swf"/>
 </mx:VBox>

 <mx:VBox id="vb2" label="Bar Chart Module">
 <mx:Label id="l2" text="BarChartModule.swf"/>
 <mx:ModuleLoader url="BarChartModule.swf"/>
 </mx:VBox>
 </mx:TabNavigator>
 </mx:Panel>
</mx:Application>
```

You can also use the `ModuleLoader` API to load and unload modules with the `loadModule()` and `unloadModule()` methods. These methods take no parameters; the `ModuleLoader` loads or unloads the module that matches the value of the current `url` property.

The following example loads and unloads the module when you click the button:

```
<?xml version="1.0"?>
<!-- modules/ASModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 import mx.modules.*;

 public function createModule(m:ModuleLoader, s:String):void {
 if (!m.url) {
 m.url = s;
 return;
 }
 m.loadModule();
 }

 public function removeModule(m:ModuleLoader):void {
 m.unloadModule();
 }
]]>
 </mx:Script>

 <mx:Panel title="Module Example"
 height="90%"
 width="90%"
 paddingTop="10"
 paddingLeft="10"
 paddingRight="10"
 paddingBottom="10"
 >
 <mx:TabNavigator id="tn"
 width="100%"
 height="100%"
 creationPolicy="auto"
 >
 <mx:VBox id="vb1" label="Column Chart Module">
 <mx:Button
 label="Load"
 click="createModule(chartModuleLoader, ll.text)"
 />
 <mx:Button
 label="Unload"
 click="removeModule(chartModuleLoader)"
 />
 <mx:Label id="ll" text="ColumnChartModule.swf"/>
 </mx:VBox>
 </mx:TabNavigator>
 </mx:Panel>
</mx:Application>
```

```

<mx:ModuleLoader id="chartModuleLoader"/>
</mx:VBox>

<mx:VBox id="vb2" label="Form Module">
 <mx:Button
 label="Load"
 click="createModule(formModuleLoader, 12.text)"
 />
 <mx:Button
 label="Unload"
 click="removeModule(formModuleLoader)"
 />
 <mx:Label id="12" text="FormModule.swf"/>
 <mx:ModuleLoader id="formModuleLoader"/>
</mx:VBox>
</mx:TabNavigator>
</mx:Panel>
</mx:Application>

```

When you load a module, Flex ensures that there is only one copy of a module loaded, no matter how many times you call the `load()` method for that module.

Modules are loaded into the child of the current application domain. You can specify a different application domain by using the `applicationDomain` property of the `ModuleLoader` class.

When two classes of the same name but different implementations are loaded, the first one loaded is the one that is used.

## Using the `ModuleManager` class to load modules

You can use the `ModuleManager` class to load the module. This technique is less abstract than using the `<mx:ModuleLoader>` tag, but it does provide you with greater control over how and when the module is loaded.

To use the `ModuleManager` to load a module, you first get a reference to the module's `IModuleInfo` interface by using the `ModuleManager`'s `getModule()` method. You then call the interface's `load()` method. Finally, you use the `factory` property of the interface to call the `create()` method, and cast the return value as the module's class.

The following example shell application loads the `ColumnChartModule.swf` file and then adds it to the display list so that it appears when the application starts:

```
<?xml version="1.0"?>
<!-- modules/ModuleLoaderApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
 <mx:Script>
 <![CDATA[
 import mx.events.ModuleEvent;
 import mx.modules.ModuleManager;
 import mx.modules.IModuleInfo;

 public var info:IModuleInfo;

 private function initApp():void {
 info = ModuleManager.getModule("BarChartModule.swf");
 info.addEventListener(ModuleEvent.READY, modEventHandler);

 // Load the module into memory. Calling load() makes the
 // IFlexModuleFactory available. You can then get an
 // instance of the class using the factory's create()
 // method.
 info.load();
 }

 private function modEventHandler(e:ModuleEvent):void {
 // Add an instance of the module's class to the
 // display list.
 vb1.addChild(info.factory.create() as BarChartModule);
 }
]]>
 </mx:Script>

 <mx:VBox id="vb1"/>
</mx:Application>
```

MXML-based modules can load other modules. Those modules can load other modules, and so on.

## Loading modules from different servers

To load a module from one server into an application running on a different server, you must establish a trust between the module and the application that loads it.

**To allow access across domains:**

1. In your loading application, you must call the `allowDomain()` method and specify the target domain from which you load a module. So, specify the target domain in the `preinitialize` event handler of your application to ensure that the application is set up before the module is loaded.
2. In the cross-domain file of the remote server where your module is, add an entry that specifies the server on which the loading application is running.
3. Load the cross-domain file on the remote server in the `preinitialize` event handler of your loading application.
4. In the loaded module, call the `allowDomain()` method so that it can communicate with the loader.

The following example shows the `init()` method of the loading application:

```
public function setup():void {
 Security.allowDomain("remoteservername");
 Security.loadPolicyFile("http://remoteservername/crossdomain.xml");
 var request:URLRequest = new URLRequest("http://remoteservername
 /crossdomain.xml");
 var loader:URLLoader = new URLLoader();
 loader.load(request);
}
```

The following example shows the loaded module's `init()` method:

```
public function initMod():void {
 Security.allowDomain("loaderservername");
}
```

The following example shows the cross-domain file that resides on the remote server:

```
<!-- crossdomain.xml file located at the root of the server -->
<cross-domain-policy>
 <allow-access-from domain="loaderservername" to-ports="*" />
</cross-domain-policy>
```

For more information about using the cross-domain policy file, see Chapter 4, “Applying Flex Security,” in *Building and Deploying Flex 2 Applications*.

## Preloading modules

When you first start an application that uses modules, the application's file size should be smaller than a similar application that does not use modules. As a result, there should be a reduction in wait time because the application can be loaded into memory and run in the Player before the modules' SWF files are even transferred across the network. However, there will be a delay when the user navigates to a part in the application that uses the module. This is because the modules are not by default preloaded, but rather loaded when they are first requested.

When a module is loaded by the Flex application for the first time, the module's SWF file is transferred across the network and stored in the browser's cache. If the Flex application unloads that module, but then later reloads it, there should be less wait time because Flash Player loads the module from the cache rather than across the network.

Module SWF files, like all SWF files, reside in the browser's cache unless and until a user clears them. As a result, modules can be loaded by the main application across several sessions, reducing load time; but this depends on how frequently the browser's cache is flushed.

You can preload modules at any time so that you can have the modules' SWF files in memory even if the module is not currently being used.

To preload modules on application startup, use the `IModuleInfo` class's `load()` method. This loads the module into memory but does not create an instance of the module.

The following example loads the BarChartModule.swf module when the application starts up, even though it will not be displayed until the user navigates to the second pane of the TabNavigator. Without preloading, the user would wait for the SWF file to be transferred across the network when they navigated to the second pane of the TabNavigator.

```
<?xml version="1.0"?>
<!-- modules/PreloadModulesApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="preloadModules()">
 <mx:Script>
 <![CDATA[
 import mx.events.ModuleEvent;
 import mx.modules.ModuleManager;
 import mx.modules.IModuleInfo;

 private function preloadModules():void {
 // Get a reference to the module's interface.
 var info:IModuleInfo =
 ModuleManager.getModule("BarChartModule.swf");
 info.addEventListener(ModuleEvent.READY, modEventHandler);

 // Load the module into memory. The module will be
 // displayed when the user navigates to the second
 // tab of the TabNavigator.
 info.load();
 }

 private function modEventHandler(e:ModuleEvent):void {
 trace("module event: " + e.type); // "ready"
 }
]]>
 </mx:Script>

 <mx:Panel
 title="Module Example"
 height="90%"
 width="90%"
 paddingTop="10"
 paddingLeft="10"
 paddingRight="10"
 paddingBottom="10"
 >

 <mx:Label width="100%" color="blue"
 text="Select the tabs to change the panel."/>

 <mx:TabNavigator id="tn"
 width="100%"
 height="100%"
 creationPolicy="auto"
 >
```

```
>
 <mx:VBox id="vb1" label="Column Chart Module">
 <mx:Label id="l1" text="ColumnChartModule.swf"/>
 <mx:ModuleLoader url="ColumnChartModule.swf"/>
 </mx:VBox>

 <mx:VBox id="vb2" label="Bar Chart Module">
 <mx:Label id="l2" text="BarChartModule.swf"/>
 <mx:ModuleLoader url="BarChartModule.swf"/>
 </mx:VBox>
</mx:TabNavigator>
</mx:Panel>
</mx:Application>
```

## Using ModuleLoader events

The `ModuleLoader` class triggers several events, including `setup`, `ready`, `loading`, `unload`, `progress`, `error`, and `urlChanged`. You can use these events to track the progress of the loading process, and find out when a module has been unloaded or when the `ModuleLoader`'s target URL has changed.

The following example uses a custom `ModuleLoader` component. This component reports all the events of the modules as they are loaded by the main application.

### Custom ModuleLoader:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- modules/CustomModuleLoader.mxml -->
<mx:ModuleLoader xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
creationComplete="init()">
 <mx:Script>
 <![CDATA[
 public function init():void {
 addEventListener("urlChanged", onUrlChanged);
 addEventListener("loading", onLoading);
 addEventListener("progress", onProgress);
 addEventListener("setup", onSetup);
 addEventListener("ready", onReady);
 addEventListener("error", onError);
 addEventListener("unload", onUnload);

 stdin = panel;
 removeChild(stdin);
 }

 public function onUrlChanged(event:Event):void {
 if (url == null) {
 if (contains(stdin))
 removeChild(stdin);
 } else {
 if (!contains(stdin))
 addChild(stdin);
 }
 progress.indeterminate=true;
 unload.enabled=false;
 reload.enabled=false;
 }

 public function onLoading(event:Event):void {
 progress.label="Loading module " + url;
 if (!contains(stdin))
 addChild(stdin);

 progress.indeterminate=true;
 unload.enabled=false;
 reload.enabled=false;
 }

 public function onProgress(event:Event):void {
 progress.label="Loaded %1 of %2 bytes...";
 progress.indeterminate=false;
 unload.enabled=true;
 reload.enabled=false;
 }
]]>
 </mx:Script>
</mx:ModuleLoader>
```

```

public function onSetup(event:Event):void {
 progress.label="Module " + url + " initialized!";
 progress.indeterminate=false;
 unload.enabled=true;
 reload.enabled=true;
}

public function onReady(event:Event):void {
 progress.label="Module " + url + " successfully loaded!";
 unload.enabled=true;
 reload.enabled=true;

 if (contains(standin))
 removeChild(standin);
}

public function onError(event:Event):void {
 progress.label="Error loading module " + url;
 unload.enabled=false;
 reload.enabled=true;
}

public function onUnload(event:Event):void {
 if (url == null) {
 if (contains(standin))
 removeChild(standin);
 } else {
 if (!contains(standin))
 addChild(standin);
 }
 progress.indeterminate=true;
 progress.label="Module " + url + " was unloaded!";
 unload.enabled=false;
 reload.enabled=true;
}

public var standin:DisplayObject;
]]>
</mx:Script>

<mx:Panel id="panel" width="100%">
 <mx:ProgressBar width="100%" id="progress" source="{this}"/>
 <mx:HBox width="100%">
 <mx:Button id="unload"
 label="Unload Module"
 click="unloadModule()"
 />
 <mx:Button id="reload"
 label="Reload Module"
 click="unloadModule();loadModule()"

```

```

 />
 </mx:HBox>
</mx:Panel>
</mx:ModuleLoader>

```

### Main application:

```

<?xml version="1.0"?>
<!-- modules/EventApp.mxml -->
<mx:Application xmlns="*" xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 [Bindable]
 public var selectedItem:Object;
]]>
 </mx:Script>
 <mx:ComboBox
 width="215"
 labelField="label"
 close="selectedItem=ComboBox(event.target).selectedItem"
 >
 <mx:dataProvider>
 <mx:Object label="Select Coverage"/>
 <mx:Object
 label="Life Insurance"
 module="insurancemodules/LifeInsurance.swf"
 />
 <mx:Object
 label="Auto Insurance"
 module="insurancemodules/AutoInsurance.swf"
 />
 <mx:Object
 label="Home Insurance"
 module="insurancemodules/HomeInsurance.swf"
 />
 </mx:dataProvider>
 </mx:ComboBox>

 <mx:Panel width="100%" height="100%">
 <CustomModuleLoader id="mod"
 width="100%"
 url="{selectedItem.module}"
 />
 </mx:Panel>
 <mx:HBox>
 <mx:Button label="Unload" click="mod.unloadModule()"/>
 <mx:Button label="Nullify" click="mod.url = null"/>
 </mx:HBox>
</mx:Application>

```

The insurance modules used in this example are simple forms, such as the following:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- modules/insurancemodules/AutoInsurance.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
background-color="#ffffff" width="100%" height="100%">
 <mx:Label
 x="147"
 y="50"
 text="Auto Insurance"
 font-size="28"
 font-family="Myriad Pro"
 />
 <mx:Form left="47" top="136">
 <mx:FormHeading label="Coverage"/>
 <mx:FormItem label="Latte Spillage">
 <mx:TextInput id="latte" width="200" />
 </mx:FormItem>
 <mx:FormItem label="Shopping Cart to the Door">
 <mx:TextInput id="cart" width="200" />
 </mx:FormItem>
 <mx:FormItem label="Irate Moose">
 <mx:TextInput id="moose" width="200" />
 </mx:FormItem>
 <mx:FormItem label="Color Fade">
 <mx:ColorPicker />
 </mx:FormItem>
 </mx:Form>
</mx:Module>
```

## Using the error event

The `error` event gives you an opportunity to gracefully fail when a module does not load for some reason. In the following example, you can load and unload a module by using the `Button` controls. To trigger an `error` event, change the URL in the `TextInput` control to a module that does not exist. The error handler displays a message to the user and writes the error message to the trace log.

```
<?xml version="1.0"?>
<!-- modules/ErrorEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 import mx.events.ModuleEvent;
 import mx.modules.*;
 import mx.controls.Alert;

 private function errorHandler(e:ModuleEvent):void {
 Alert.show("There was an error loading the module." +
```

```

 " Please contact the Help Desk.");
 trace(e.errorText);
 }

 public function createModule():void {
 if (chartModuleLoader.url == til.text) {
 // If they are the same, call loadModule.
 chartModuleLoader.loadModule();
 } else {
 // If they are not the same, then change the url,
 // which triggers a call to the loadModule() method.
 chartModuleLoader.url = til.text;
 }
 }

 public function removeModule():void {
 chartModuleLoader.unloadModule();
 }

]]]>
</mx:Script>

<mx:Panel title="Module Example"
 height="90%"
 width="90%"
 paddingTop="10"
 paddingLeft="10"
 paddingRight="10"
 paddingBottom="10"
>
 <mx:HBox>
 <mx:Label text="URL:"/>
 <mx:TextInput width="200" id="til" text="ColumnChartModule.swf"/>
 </mx:HBox>
 <mx:Button label="Load" click="createModule()"/>
 <mx:Button label="Unload" click="removeModule()"/>
 <mx:ModuleLoader id="chartModuleLoader" error="errorHandler(event)"/>
</mx:Panel>
</mx:Application>

```

## Using the progress event

You can use the `progress` event to track the progress of a module as it loads. When you add a listener for the `progress` event, Flex calls that listener at regular intervals during the module's loading process. Each time the listener is called, you can look at the `bytesLoaded` property of the event. You can compare this to the `bytesTotal` property to get a percentage of completion.

The following example reports the level of completion during the module's loading process. It also produces a simple progress bar that shows users how close the loading is to being complete.

```
<?xml version="1.0"?>
<!-- modules/SimpleProgressEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 import mx.events.ModuleEvent;
 import flash.events.ProgressEvent;
 import mx.modules.*;

 [Bindable]
 public var progBar:String = "";
 [Bindable]
 public var progMessage:String = "";

 private function progressEventHandler(e:ProgressEvent):void {
 progBar += ".";
 progMessage =
 "Module " +
 Math.round((e.bytesLoaded/e.bytesTotal) * 100) +
 "% loaded";
 }

 public function createModule():void {
 chartModuleLoader.loadModule();
 }

 public function removeModule():void {
 chartModuleLoader.unloadModule();
 progBar = "";
 progMessage = "";
 }
]]>
 </mx:Script>

 <mx:Panel title="Module Example"
 height="90%"
 width="90%"
```

```
paddingTop="10"
paddingLeft="10"
paddingRight="10"
paddingBottom="10"
>
<mx:HBox>
 <mx:Label id="l2" text="{progMessage}"/>
 <mx:Label id="l1" text="{progBar}"/>
</mx:HBox>

<mx:Button label="Load" click="createModule()"/>
<mx:Button label="Unload" click="removeModule()"/>

<mx:ModuleLoader
 id="chartModuleLoader"
 url="ColumnChartModule.swf"
 progress="progressEventHandler(event)"
/>
</mx:Panel>
</mx:Application>
```

You can also connect a module loader to a `ProgressBar` control. The following example creates a custom component for the `ModuleLoader` that includes a `ProgressBar` control. The `ProgressBar` control displays the progress of the module loading.

```
<?xml version="1.0"?>
<!-- modules/MySimpleModuleLoader.mxml -->
<mx:ModuleLoader xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 private function clickHandler():void {
 if (!url) {
 url="ColumnChartModule.swf";
 }
 loadModule();
 }
]]>
 </mx:Script>

 <mx:ProgressBar
 id="progress"
 width="100%"
 source="{this}"
 />
 <mx:HBox width="100%">
 <mx:Button
 id="load"
 label="Load"
 click="clickHandler()"
 />
 <mx:Button
 id="unload"
 label="Unload"
 click="unloadModule()"
 />
 <mx:Button
 id="reload"
 label="Reload"
 click="unloadModule();loadModule();"
 />
 </mx:HBox>
</mx:ModuleLoader>
```

You can use this module in a simple application, as the following example shows:

```
<?xml version="1.0"?>
<!-- modules/ComplexProgressEventHandler.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:local="*">

 <mx:Panel title="Module Example"
 height="90%"
 width="90%"
 paddingTop="10"
 paddingLeft="10"
 paddingRight="10"
 paddingBottom="10"
 >
 <mx:Label text="Use the buttons below to load and unload
 the module."/>
 <local:MySimpleModuleLoader id="customLoader"/>
 </mx:Panel>
</mx:Application>
```

This example does not change the `ProgressBar`'s `label` property for all events. For example, if you load and then unload the module, the `label` property remains at "LOADING 100%". To adjust the label properly, you must define other event handlers for the `ModuleLoader` events, such as `unload` and `error`.

## Passing data

Communication between modules and the parent application, and among modules, is possible. You can use the following approaches to facilitate inter-module, application-to-module, and module-to-application communication:

- `ModuleLoader`'s `child`, `ModuleManager`'s `factory`, and `Application`'s `parentApplication` properties — You can use these properties to access modules and applications. However, by using these properties, you might create a tightly-coupled design that prevents code reuse. In addition, you might also create dependencies among modules and applications that cause class sizes to be bigger. For more information, see “Accessing modules from the parent application” on page 1129, “Accessing the parent application from the modules” on page 1132, and “Accessing modules from other modules” on page 1134.
- Query string parameters — Modules are loaded with a URL; you can pass parameters on this URL and then parse those parameters in the module. For more information, see “Passing data with the query string” on page 1136.

- Interfaces — You can create ActionScript interfaces that define the methods and properties that modules and applications can access. This gives you greater control over module and application interaction. It also prevents you from creating dependencies between modules and applications. For more information, see “Using interfaces for module communication” on page 1139.

The techniques for accessing methods and properties described in this section apply to parent applications as well as modules. Modules can load other modules, which makes the loading module somewhat like the parent application in the simpler examples.

## Accessing modules from the parent application

You can access the methods and properties of a module from its parent application. To do this, you must get an instance of the module’s class.

If you use the `ModuleLoader` to load the module, you can call methods on a module from the parent application by referencing the `ModuleLoader`’s `child` property, and casting it to the module’s class. The `child` property is an instance of the module’s class. In this case, the module’s class is the name of the MXML file that defines the module.

The following example calls the module’s `getTitle()` method from the parent application:

**Parent Application:**

```
<?xml version="1.0"?>
<!-- modules/ParentApplication.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script><![CDATA[
 [Bindable]
 private var s:String;

 private function getTitle():void {
 s = (m1.child as ChildModule1).getModTitle();
 }
]]></mx:Script>
 <mx:Label id="l1" text="{s}"/>
 <mx:ModuleLoader url="ChildModule1.swf" id="m1" ready="getTitle()"/>
</mx:Application>
```

### Module:

```
<?xml version="1.0"?>
<!-- modules/ChildModule1.xml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%">
 <mx:Script><![CDATA[
 // Defines the method that the application calls.
 public function getModTitle():String {
 return "Child Module 1";
 }
]]></mx:Script>
</mx:Module>
```

This approach creates a tight coupling between the application and the module, and does not easily let you use the same code when loading multiple modules. Another technique is to use an interface to define the methods and properties on the module (or group of modules) that the application can call. For more information, see “Using interfaces for module communication” on page 1139.

If you load the module that you want to call by using the ModuleManager API, there is some additional coding in the shell application. You use the ModuleManager’s `factory` property to get an instance of the module’s class. You can then call the module’s method on that instance.

The following module example defines a single method, `computeAnswer()`:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/SimpleModule.xml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 public function computeAnswer(a:Number, b:Number):Number {
 return a + b;
 }
]]>
 </mx:Script>
</mx:Module>
```

The following example gets an instance of the SimpleModule class by using the factory property to call the create() method. It then calls the computeAnswer() method on that instance:

```
<?xml version="1.0"?>
<!-- modules/mxmlmodules/SimpleMXMLApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
 <mx:Script>
 <![CDATA[
 import mx.modules.IModuleInfo;
 import mx.modules.ModuleManager;

 public var assetModule:IModuleInfo;
 public var sm:Object;

 [Bindable]
 public var answer:Number = 0;

 public function initApp():void {
 // Get the IModuleInfo interface for the specified URL.
 assetModule = ModuleManager.getModule("SimpleModule.swf");
 assetModule.addEventListener("ready", getModuleInstance);
 assetModule.load();
 }

 public function getModuleInstance(e:Event):void {
 // Get an instance of the module.
 sm = assetModule.factory.create() as SimpleModule;
 }

 public function addNumbers():void {
 var a:Number = Number(ti1.text);
 var b:Number = Number(ti2.text);

 // Call a method on the module.
 answer = sm.computeAnswer(a, b).toString();
 }
]]>
 </mx:Script>

 <mx:Form>
 <mx:FormHeading label="Enter values to sum."/>
 <mx:FormItem label="First Number">
 <mx:TextInput id="ti1" width="50"/>
 </mx:FormItem>
 <mx:FormItem label="Second Number">
 <mx:TextInput id="ti2" width="50"/>
 </mx:FormItem>
 <mx:FormItem label="Result">
```

```
 <mx:Label id="ti3" width="100" text="{answer}"/>
 </mx:FormItem>
 <mx:Button id="b1" label="Compute" click="addNumbers()"/>
</mx:Form>
</mx:Application>
```

In this example, you should actually create a module that extends the `ModuleBase` class in `ActionScript` rather than an `MXML`-based module that extends the `Module` class. This is because this example does not have any visual elements and contains only a single method that computes and returns a value. A module that extends the `ModuleBase` class would be more lightweight than a class that extends `Module`. For more information on writing `ActionScript`-based modules that extend the `ModuleBase` class, see “Creating `ActionScript`-based modules” on page 1101.

## Accessing the parent application from the modules

Modules can access properties and methods of the parent application by using a reference to the `parentApplication` property.

The following example accesses the `expenses` property of the parent application when the module first loads. The module then uses this property, an `ArrayCollection`, as the source for its chart's data. When the user clicks the button, the module calls the `getNewData()` method of the parent application that returns a new `ArrayCollection` for the chart:

```
<?xml version="1.0"?>
<!-- modules/ChartChildModule.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" creationComplete="getDataFromParent()">
 <mx:Script><![CDATA[
 import mx.collections.ArrayCollection;

 [Bindable]
 private var expenses:ArrayCollection;

 // Access properties of the parent application.
 private function getDataFromParent():void {
 expenses = parentApplication.expenses;
 }
]]></mx:Script>
 <mx:ColumnChart id="myChart" dataProvider="{expenses}">
 <mx:horizontalAxis>
 <mx:CategoryAxis
 dataProvider="{expenses}"
 categoryField="Month"
 />
 </mx:horizontalAxis>
 <mx:series>
 <mx:ColumnSeries
 xField="Month"
 yField="Profit"
 displayName="Profit"
 />
 <mx:ColumnSeries
 xField="Month"
 yField="Expenses"
 displayName="Expenses"
 />
 </mx:series>
 </mx:ColumnChart>
 <mx:Legend dataProvider="{myChart}"/>

 <mx:Button id="b1" click="expenses = parentApplication.getNewData();"
label="Get New Data"/>
</mx:Module>
```

The following example shows the parent application that the previous example module uses:

```
<?xml version="1.0"?>
<!-- modules/ChartChildModuleLoader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script><![CDATA[
 import mx.collections.ArrayCollection;

 [Bindable]
 public var expenses:ArrayCollection = new ArrayCollection([
 {Month:"Jan", Profit:2000, Expenses:1500},
 {Month:"Feb", Profit:1000, Expenses:200},
 {Month:"Mar", Profit:1500, Expenses:500}
]);

 public function getNewData():ArrayCollection {
 return new ArrayCollection([
 {Month:"Apr", Profit:1000, Expenses:1100},
 {Month:"May", Profit:1300, Expenses:500},
 {Month:"Jun", Profit:1200, Expenses:600}
]);
 }
]]></mx:Script>
 <mx:ModuleLoader url="ChartChildModule.swf" id="m1"/>
</mx:Application>
```

You can also call methods and access properties on other modules. For more information, see “Accessing modules from other modules” on page 1134.

The drawbacks to this approach is that it can create dependencies on the parent application inside the module. In addition, the modules are no longer portable across multiple applications unless you ensure that you replicate the behavior of the applications.

To avoid these drawbacks, you should use interfaces that secure a contract between the application and its modules. This contract defines the methods and properties that you can access. Having an interface lets you reuse the application and modules as long as you keep the interface updated. For more information, see “Using interfaces for module communication” on page 1139.

## Accessing modules from other modules

You can access properties and methods of other modules using references to the other modules through the parent application. You do this by using the `ModuleLoader`’s `child` property. This property points to an instance of the module’s class, which lets you call methods and access properties.

The following example defines a single application that loads two modules. The InterModule1 module defines a method that returns a String. The InterModule2 module calls that method and sets the value of its Label to the return value of that method.

**Main application:**

```
<?xml version="1.0"?>
<!-- modules/TitleModuleLoader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script><![CDATA[

]]></mx:Script>
 <mx:ModuleLoader url="InterModule1.swf" id="m1"/>
 <mx:ModuleLoader url="InterModule2.swf" id="m2"/>
</mx:Application>
```

**Module 1:**

```
<?xml version="1.0"?>
<!-- modules/InterModule1.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%">
 <mx:Script><![CDATA[
 // Defines the method that the other module calls.
 public function getNewTitle():String {
 return "New Module Title";
 }
]]></mx:Script>
</mx:Module>
```

**Module 2:**

```
<?xml version="1.0"?>
<!-- modules/InterModule2.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%">
 <mx:Script><![CDATA[
 [Bindable]
 private var title:String;

 // Call method of another module.
 private function changeTitle():void {
 title = parentApplication.m1.child.getNewTitle();
 }

]]></mx:Script>
 <mx:HBox>
 <mx:Label id="l1" text="Title: "/>
 <mx:Label id="myTitle" text="{title}"/>
 </mx:HBox>
 <mx:Button id="b1" label="Change Title" click="changeTitle()"/>
</mx:Module>
```

The application in this example lets the two modules communicate with each other. You could, however, define methods and properties on the application that the modules could access. For more information, see “Accessing the parent application from the modules” on page 1132.

As with accessing the parent application’s properties and methods directly (described in “Accessing the parent application from the modules” on page 1132), using the technique described in this section can make your modules difficult to reuse and can also create dependencies that can cause the module to be larger than necessary. You should instead try to use interfaces to define the contract between modules. For more information, see “Using interfaces for module communication” on page 1139.

## Passing data with the query string

One way to pass data to a module is to append query string parameters to the URL that you use to load the module from. You can then parse the query string by using ActionScript to access the data.

In the module, you can access the URL by using the `loaderInfo` property. This property points to the `flash.display.LoaderInfo` object of the loading SWF (in this case, the shell application). The information provided by the `LoaderInfo` object includes load progress, the URLs of the loader and loaded content, the file size of the application, and the height and width of the application.

The following example application builds a unique query string for the module that it loads. The query string includes a `firstName` and `lastName` parameter.

```
<?xml version="1.0"?>
<!-- modules/QueryStringApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="500"
width="400">
 <mx:Script><![CDATA[

 public function initModule():void {
 // Build query string so that it looks something like this:
 // "QueryStringModule.swf?firstName=Nick&lastName=Danger"
 var s:String = "QueryStringModule.swf?" + "firstName=" +
 ti1.text + "&lastName=" + ti2.text;

 // Changing the url property of the ModuleLoader causes
 // the ModuleLoader to load a new module.
 m1.url = s;
 }

]]></mx:Script>

 <mx:Form>
 <mx:FormItem id="fi1" label="First Name:">
 <mx:TextInput id="ti1"/>
 </mx:FormItem>
 <mx:FormItem id="fi2" label="Last Name:">
 <mx:TextInput id="ti2"/>
 </mx:FormItem>
 </mx:Form>

 <mx:ModuleLoader id="m1"/>

 <mx:Button id="b1" label="Submit" click="initModule()"/>
</mx:Application>
```

The following example module parses the query string that was used to load it. If the `firstName` and `lastName` parameters are set, the module prints the results in a `TextArea`. The module also traces some additional information available through the `LoaderInfo` object:

```
<?xml version="1.0"?>
<!-- modules/QueryStringModule.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="parseString()">
 <mx:Script>
 <![CDATA[
 import mx.utils.*;

 [Bindable]
 private var salutation:String;

 public var o:Object = {};

 public function parseString():void {
 try {
 // Remove everything before the question mark, including
 // the question mark.
 var myPattern:RegExp = /.*\?/;
 var s:String = this.loaderInfo.url.toString();
 s = s.replace(myPattern, "");

 // Create an Array of name=value Strings.
 var params:Array = s.split("&");

 // Print the params that are in the Array.
 var keyStr:String;
 var valueStr:String;
 var paramObj:Object = params;
 for (keyStr in paramObj) {
 valueStr = String(paramObj[keyStr]);
 tal.text += keyStr + ":" + valueStr + "\n";
 }

 // Set the values of the salutation.
 for (var i:int = 0; i < params.length; i++) {
 var tempA:Array = params[i].split("=");
 if (tempA[0] == "firstName") {
 o.firstName = tempA[1];
 }
 if (tempA[0] == "lastName") {
 o.lastName = tempA[1];
 }
 }
 if (StringUtil.trim(o.firstName) != "" &&
 StringUtil.trim(o.lastName) != "") {
 salutation = "Welcome " +

```

```

 o.firstName + " " + o.lastName + "!";
 } else {
 salutation = "Full name not entered."
 }
} catch (e:Error) {
 trace(e);
}

// Show some of the information available through loaderInfo:
trace("AS version: " + this.loaderInfo.actionScriptVersion);
trace("App height: " + this.loaderInfo.height);
trace("App width: " + this.loaderInfo.width);
trace("App bytes: " + this.loaderInfo.bytesTotal);
}
]]>
</mx:Script>
<mx:Label text="{salutation}"/>
<mx:TextArea height="100" width="300" id="ta1"/>
</mx:Module>

```

Modules are cached by their URL, including the query string. As a result, you will load a new module if you change the URL or any of the query string parameters on the URL. This can be useful if you want multiple instances of a module based on the parameters that you pass in the URL to the module loaded.

## Using interfaces for module communication

You can use an interface to provide module-to-application communication. Your modules implement the interface and your application calls its methods or sets its properties. The interface defines stubs for the methods and properties that you want the application and module to share. The module implements an interface known to the application, or the application implements an interface known to the module. This lets you avoid so-called hard dependencies between the module and the application.

In the main application, when you want to call methods on the module, you cast the `ModuleLoader`'s `child` property to an instance of the custom interface.

The following example application lets you customize the appearance of the module that it loads by calling methods on the `IModuleInterface` interface. The application also calls the `getModuleName()` method. This method returns a value from the module, and sets a local property to that value.

```
<?xml version="1.0"?>
<!-- modules/interfaceexample/Main.mxml -->
<mx:Application xmlns="*" xmlns:mx="http://www.adobe.com/2006/mxml">
 <mx:Script>
 <![CDATA[
 import mx.events.ModuleEvent;
 import mx.modules.ModuleManager;

 [Bindable]
 public var selectedItem:Object;

 [Bindable]
 public var currentModuleName:String;

 private function applyModuleSettings(e:Event):void {
 // Cast the ModuleLoader's child to the interface.
 // This child is an instance of the module.
 // You can now call methods on that instance.
 var ichild:* = mod.child as IModuleInterface;
 if (mod.child != null) {
 // Call setters in the module to adjust its
 // appearance when it loads.
 ichild.setAdjusterID(myId.text);
 ichild.setBackgroundColor(myColor.selectedColor);
 } else {
 trace("Uh oh. The mod.child property is null");
 }
 // Set the value of a local variable by calling a method
 // on the interface.
 currentModuleName = ichild.getModuleName();
 }

 private function reloadModule():void {
 mod.unloadModule();
 mod.loadModule();
 }
]]>
 </mx:Script>

 <mx:Form>
 <mx:FormItem label="Current Module:">
 <mx:Label id="l1" text="{currentModuleName}"/>
 </mx:FormItem>
 <mx:FormItem label="Adjuster ID:">
 <mx:TextInput id="myId" text="Enter your ID"/>
 </mx:FormItem>
 </mx:Form>
</mx:Application>
```

```

 </mx:FormItem>
 <mx:FormItem label="Background Color:">
 <mx:ColorPicker id="myColor"
 selectedColor="0xFFFFFF"
 change="reloadModule()"
 />
 </mx:FormItem>
 </mx:Form>

 <mx:Label text="Long Shot Insurance" fontSize="24"/>
 <mx:ComboBox
 labelField="label"
 close="selectedItem=ComboBox(event.target).selectedItem"
 >
 <mx:dataProvider>
 <mx:Object label="Select Module"/>
 <mx:Object label="Auto Insurance" module="AutoInsurance.swf"/>
 </mx:dataProvider>
 </mx:ComboBox>

 <mx:Panel width="100%" height="100%">
 <mx:ModuleLoader id="mod"
 width="100%"
 url="{selectedItem.module}"
 ready="applyModuleSettings(event)"
 />
 </mx:Panel>
 <mx:Button id="b1" label="Reload Module" click="reloadModule()"/>
</mx:Application>

```

The following example defines a simple interface that has two getters and one setter. This interface is used by the application in the previous example.

```

// modules/interfaceexample/IModuleInterface
package
{
 import flash.events.IEventDispatcher;

 public interface IModuleInterface extends IEventDispatcher {

 function getModuleName():String;
 function setAdjusterID(s:String):void;
 function setBackgroundColor(n:Number):void;
 }
}

```

The following example defines the module that is loaded by the previous example. It implements the custom `IModuleInterface` interface.

```
<?xml version="1.0"?>
<!-- modules/interfaceexample/AutoInsurance.mxml -->
<mx:Module xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" implements="IModuleInterface">

 <mx:Panel id="p1"
 title="Auto Insurance"
 width="100%"
 height="100%"
 backgroundColor="{bgcolor}"
 >
 <mx:Label id="myLabel" text="ID: {adjuster}"/>
 </mx:Panel>

 <mx:Script>
 <![CDATA[
 [Bindable]
 private var adjuster:String;
 [Bindable]
 private var bgcolor:Number;

 public function setAdjusterID(s:String):void {
 adjuster = s;
 }

 public function setBackgroundColor(n:Number):void {
 // Use a bindable property to set values of controls
 // in the module. This ensures that the property will be set
 // even if Flex applies the property after the module is
 // loaded but before it is rendered by the player.
 bgcolor = n;

 // Don't do this. The backgroundColor style might not be set
 // by the time the ModuleLoader triggers the READY
 // event:
 // p1.setStyle("backgroundColor", n);
 }

 public function getModuleName():String {
 return "Auto Insurance";
 }
]]>
 </mx:Script>
</mx:Module>
```

In general, if you want to set properties on controls in the module by using external values, you should create variables that are bindable. You then set the values of those variables in the interface's implemented methods. If you try to set properties of the module's controls directly by using external values, the controls might not be instantiated by the time the module is loaded and the attempt to set the properties might fail.

