InterSystems
Caché

Using InterSystems Development Environments — Atelier and Studio

Version 2017.2
2020-06-25

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using InterSystems Development Environments — Atelier and Studio
Caché Version 2017.2 2020-06-25

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Figures

List of Tables

About This Book

This book provides information on the integrated development environments (IDEs) available from InterSystems. It contains the following chapters and appendices:

- Introduction to Atelier — For information on using Atelier, see http://www.intersystems.com/atelier, the Atelier home page.
- Introduction to Studio — This chapter and the remaining chapters apply to Studio.
- Building a Simple Application with Studio
- Creating Class Definitions
- Adding Properties to a Class
- Adding Methods to a Class
- Adding Class Parameters to a Class
- Adding Relationships to a Class
- Adding Queries to a Class
- Adding Indices to a Class
- Adding Projections to a Class
- Adding XData Blocks to a Class
- Adding SQL Triggers and Foreign Keys
- Adding Storage Definitions to a Class
- Working with CSP Files
- Working with Routines
- Using the Studio Debugger
- Using Studio Templates
- Studio Menu Reference
- Setting Studio Options
- Using Studio Source Control Hooks
- Frequently Asked Questions About Studio
- And a more detailed Table of Contents.

Also see Class Definitions in the reference Class Definitions in *Caché Class Definition Reference*.

1

Introduction to Atelier

With the release of version 2016.2, InterSystems introduces Atelier, a new IDE based on the open-source Eclipse development environment. Atelier is in active development and will provide a full Caché development environment in the future. Studio remains available for customers with experience and investment in that technology.

Atelier is available as a separate download in addition to Caché or Ensemble. You can choose to install either a stand-alone Rich Client Platform (RCP) application, or a plug-in that can be added to an existing Eclipse installation. Users of the RCP application can add additional Eclipse plug-ins. Atelier uses the Eclipse auto-update mechanism to help users get the latest changes. For more information on Atelier, see http://www.intersystems.com/atelier, the Atelier home page, which has information on downloading and using Atelier. (The remaining chapters of this book describe Studio.)

2

Introduction to Studio

Studio offers features that help you develop applications rapidly, in a single, integrated environment including:

- An editor in which to create
  - Classes, including persistent, database classes and Web service classes
  - Interactive Web pages using: CSP (Caché Server Pages) and Zen, XML, HTML, JavaScript, cascading style sheets (CSS)
  - Routines using ObjectScript, Basic, or MultiValue
- Integrated syntax coloring and syntax checking for ObjectScript, Basic, Java, SQL, JavaScript, HTML, and XML.
- Support for teams of developers working with a common repository of application source code.
- A graphical source code debugger.
- The ability to organize application source code into projects.

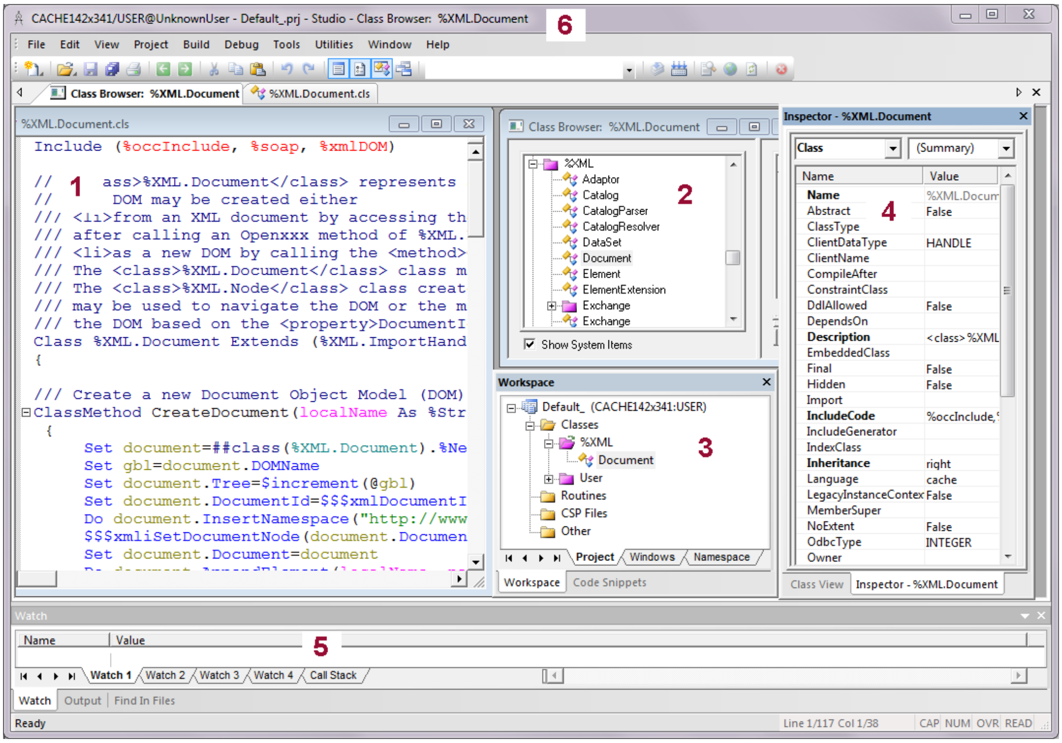
Studio is a client application, built using Caché objects, that runs on Windows-based operating systems. It can connect to any Caché server (compatible with the current version of Studio) regardless of what platform and operating system that server is using.

Note: A Studio client must be running either the same version of Caché or a higher version than the Caché server that it is connecting to. Example: Caché 2015.1 Studio can connect to a Caché 2015.1 (or earlier version) server. Caché 2014.1 Studio cannot connect to a Caché 2015.1 (or later) server. This applies also to maintenance releases. Example: Caché 2014.1.2 Studio can connect to a Caché 2014.1.1 (or earlier maintenance release or version) server. Caché 2014.1.0 Studio cannot connect to a Caché 2014.1.1 (or later maintenance release or version) server.

2.1 Overview of the Studio Window

Studio is a standard Windows application. It uses windows to display and allow the editing of aspects. The main components of the Studio user interface are shown below:

Figure 2–1: Studio Components



```
Include (%occInclude, %soap, %xmlDOM)

//      ass>%XML.Document</class> represents
//        DOM may be created either
/// <li>from an XML document by accessing th
/// after calling an Openxxx method of %XML.
/// <li>as a new DOM by calling the <method>
/// The <class>%XML.Document</class> class m
/// The <class>%XML.Node</class> class creat
/// may be used to navigate the DOM or the m
/// the DOM based on the <property>DocumentI
Class %XML.Document Extends (%XML.ImportHand
{

/// Create a new Document Object Model (DOM)
ClassMethod CreateDocument(localName As %Str
 {
     Set document=##class(%XML.Document).%Ne
     Set gbl=document.DOMName
     Set document.Tree=$increment(@gbl)
     Set document.DocumentId=$$$xmlDocumentI
     Do document.InsertNamespace("http://www
     $$$xmliSetDocumentNode(document.Documen
     Set document.Document=document
```

Name	Value
Name	%XML.Docum
Abstract	False
ClassType	
ClientDataType	HANDLE
ClientName	
CompileAfter	
ConstraintClass	
DdlAllowed	False
DependsOn	
Description	<class> %XML
EmbeddedClass	
Final	False
Hidden	False
Import	
IncludeCode	%occInclude,
IncludeGenerator	
IndexClass	
Inheritance	right
Language	cache
LegacyInstanceContex	False
MemberSuper	
NoExtent	False
OdbcType	INTEGER
Owner	

1. *Editors: Class Editor* for editing class definitions, *Routine Editor* for editing routines and include files, and *CSP Editor* for editing CSP definition text.
2. *Class Browser window:* for viewing existing classes.
3. *Workspace window:* three tabs let you display: the contents of the current project, all open windows, or the content of the current namespace.
4. *Class Inspector window:* for viewing and modifying keywords in a class definition.
5. *Watch window:* displays variables.
6. *Title Bar:* displays *ConnectionName/Namespace@UserName - ProjectName.prj – Studio – ActiveDocument*. If the active document is maximized, the name shows in square brackets.

In addition to the windows displayed above, Studio contains wizards and templates for assisting with common tasks. These include:

- *Find in Files window:* displays a search window.
- *Output window:* displays output from the Caché server (such as messages generated during class compilation).
- *Code Snippets window:* for viewing and dragging user-created code snippets.
- *New Class wizard:* defines a new class.
- *Class member wizards that add members to class definitions for*: properties, indexes, relationships, methods, parameters, SQL triggers, queries, projections, storage, foreign keys, and XData blocks.
- *Wizards that create classes from other technologies; from*: Java classes and jar files, SML schema, SOAP client classes, that provide access to COM objects, and DLL assembly files from .NET.
- *HTML templates that add:* colors, tables, tags, and scripts.
- *CSP Form wizard*: creates an HTML form bound to a Caché object in a CSP page.

- *Zen templates that add:* charts, tables, methods, and styles.

2.1.1 Running Studio from the Command Line

You can run Studio from the system's command line using the command **Cstudio.exe** (in the Caché bin directory). The command and its parameters are case-sensitive.

Parameter	Description
?	Help info
/Servername=ServerName	Connect to the server named *ServerName*.
/Server=cn_iptcp:127.0.0.1[1972]::	Connect to the server at *ip address*[*port*].
/Namespace=User	Connect to the User namespace. You must also define a server.
/Project=MyProject	Open project *MyProject*. You must also define a server and a namespace.
cn_iptcp://127.0.0.1:1972/User/test.int	Load routine **test.int**. `cn_iptcp` is a case-sensitive protocol identifier.
/files="tag+1^myroutine.int",User.Class1.cls	Open listed documents and set cursor in specified position. You must also define a server and a namespace.
/pid=123	Attach to process. You must also define a server and a namespace.
/fastconnect=127.0.0.1[1972]:USER:_SYSTEM:SYS	Connect without connection definition in registry using *ip address*[*port*]:USER:*username*:*password*

2.2 Projects

Studio uses *projects* to organize application source code.

A project is a set of class definitions, CSP files, routines, and include files. For example, you might create a Studio project to group all classes and CSP files for a single application.

You are *always* in a project, either one that you created or the default project that is created when you first open Studio called `Default_`*yourusername* (a prefix of `Default_` followed by your username).

All files in a single project must be in the same namespace (and Caché server). Each class, CSP file, or routine can be associated with any number of projects. Each namespace can contain any number of projects.

The project stores information such as the class hierarchy in a given Caché namespace, used when you edit classes or CSP files. The project also stores debugging information (such as how to start the application you want to debug).

2.3 Class Definitions

A *class definition* defines a Caché class. A class definition consists of class members (such as properties and methods) and other items, called *keywords*, each with associated values, that specify details of the class behavior.

Class definitions reside in a Caché database where they are stored in the class dictionary. A class definition can be compiled, a process which creates executable code that can create object instances based on the class definition. The source code for the executable code created for a class consists of one or more Caché routines. These generated routines can also be viewed in Studio.

A class definition can be projected for use by other technologies. In the case of SQL and ActiveX, this projection is automatic. In the case of Java (or C++) there is an additional compilation step in which a Java class is generated that corresponds to the Caché class definition. For details, see the chapter "Adding Class Projections."

Within Studio, class definitions can be displayed and edited in a Class Editor window. Class definitions can also be viewed in the Class Inspector window as keywords and their corresponding values in tables.

2.3.1 Class Definitions as Text

The following is an example of a class definition that defines a class containing one property and one method:

```
/// Definition of a simple persistent Person class
Class MyApp.Person Extends %Persistent
{

/// The name of the Person
Property Name As %String;


/// A simple Print method
Method Print() As %Boolean
{
    // Write out Person's Name
    Write "Name is: ", ..Name
    Quit 1
}

}
```

2.3.1.1 Class Information

A class definition starts with the declaration of the class name and any class-wide characteristics, such as:

```
Class MyApp.Student Extends Person
{

}
```

This example defines a class called MyApp.Student (with no properties, methods, or other members) that extends (is derived from) the class MyApp.Person (since Student and Person are in the same package, we can omit `MyApp` from the Extends statement). The { } (braces) enclose the definition of the class members (of which there are none in this example).

You can specify additional characteristics for this class by defining values for class keywords. This is done by placing a list of keywords (possibly with values) in [ ] (brackets) immediately following the class declaration (after the class name and superclass name (if any)).

For example, you can specify that the class as `Final` and the name of its corresponding SQL table as `StudentTable`.

```
Class MyApp.Student Extends Person [Final, SqlTableName=StudentTable]
{

}
```

You can also provide a description for this class by placing a description comment (identified by ///, three slashes) immediately before the declaration of the class. This description is used when you view the class documentation via the Caché online class reference. It may contain HTML markup. Example:

```
/// This is a simple Student class
/// It is derived from the Person class
Class MyApp.Student Extends Person
{

}
```

You can use the C-style //, two slashes, and /* */, begin with slash asterisk and end with asterisk slash, comments anywhere in a class definition to comment out a section of the class definition.

2.3.1.2 Properties

You can define a property in a class definition using the `Property` keyword:

```
Class MyApp.Student Extends Person
{
     Property GPA As %Float;
}
```

This example defines a property named *GPA* with type `%Float` (specified using `As`). The end of the property definition is marked with a final semicolon (;).

As with the class declaration, you can add a description for this property using a preceding /// comment and we can specify additional property keywords in [ ] brackets:

```
Class MyApp.Student Extends Person
{
    /// Grade Point Average for the Student
    Property GPA As %Float [ Required ];
}
```

If you want to specify parameter values for the property data type (parameters give you a way to customize the behavior of a property), placed them in ( ), parentheses, as part of the type name. Note that the values for data type parameters are treated as literal values; enclose strings in quotation marks.

```
Class MyApp.Student Extends Person
{
    /// Grade Point Average for the Student
    Property GPA As %Float(MINVAL=0.0, MAXVAL=5.0) [ Required ];
}
```

2.3.1.3 Methods

You can define a method in a class definition using the `Method` keyword:

```
Class MyApp.Student Extends Person
{
/// This method wastes count seconds.
Method WasteTime(count As %Integer=1) As %Boolean [Final]
{
    // loop and sleep
    For i = 1:1:count {
        Write "."
        Hang 1
    }
    Quit 1
}
}
```

The return type of the method is specified using `As` followed by a class name. The formal argument list follows the method name and is enclosed in ( ) (parentheses). The implementation of the method is contained in { } (braces) following the method declaration.

As with a property, you can use /// (three slashes) comments to specify a description (with HTML markup, if desired) and additional keyword values are placed in [ ] (brackets).

You can specify the programming language for a method using the Language keyword. For example, the code below defines a method in Basic:

```
/// Find the sum of numbers from 1 to <var>count</var>.
Method SumUp(count As %Integer) As %Integer [Language = basic]
{
    total = 0
    For i = 1 To count
        total = total + i
    Next

    Return i
}
```

Use the `Language` keyword to specify one of the following languages:

- `cache`—ObjectScript (the default if no language is specified). Refer to the book *Using Caché ObjectScript* for more details.
- `basic`—Basic. Caché supports a variant of the BASIC programming language. Basic methods are compiled into executable code that runs in the Caché virtual machine (in the same manner as ObjectScript). Refer to *Using Basic* for more details.
- `java`—Java. When you use the Caché Java Binding, Java methods become part of the automatically generated Java classes and are compiled into executable Java code. Refer to *Using Java with Caché* for more details.

A single class can contain methods that use different languages. Or you can specify the default programming language for an entire class using the class-level Language keyword.

2.4 CSP Files

A CSP (Caché Server Page) file is an HTML or XML text file containing CSP markup language. The CSP engine processes a CSP file and generates from it a Caché class which is then used to respond to HTTP events and provide Web content.

If you prefer a more programmatic approach to Web development, you can also use Studio to create and edit CSP classes in the same way as you would work with any other class definitions.

Studio displays CSP files in a CSP Editor window. This editor provides syntax coloring of HTML and XML as well as many of the scripting languages that may be contained in a CSP file.

The CSP Editor provides commands for performing common CSP and HTML editing tasks such as inserting CSP markup tags. Studio also lets you view the results of a CSP file in a browser using the menu pick **View > Web Page**.

2.5 Routine Editor

Using the Routine Editor, you can directly create and edit the source for specific Caché routines in a syntax-coloring editor. You also use the Routine Editor to edit include files.

2.6 Multiple User Support

Studio is an object-based, client/server application. The source files—class definitions, routines, include files, and CSP files—that you can create and edit with Studio are stored in a Caché server and are represented as objects.

When you save a source file from Studio, it is saved in the Caché server you are connected to. If a source file is modified on the server while you are viewing it in Studio, you are notified and asked if you want to load the newer version.

Studio automatically detects when multiple users view the same source components simultaneously and manages access concurrency. If you attempt to open a file that is being edited by another user, you are notified and asked if you want to open the file in read-only mode.

2.7 Importing and Exporting Caché Documents Locally

Normally any documents you work with in Studio (such as class definitions or routines) are stored in a Caché database (which may be on a remote machine). You can import from and export to local files using **Tools > Export** and **Tools > Import**.

Class definitions and routines are stored in local files as XML documents.

2.8 Debugging

Studio includes a source-level, GUI debugger. The debugger attaches (or starts up and attaches to) a target process running on the same Caché server that Studio is connected to. The debugger controls this target process remotely and allows you to watch variables, step through code, and set breakpoints.

You typically must have a project open in order to use the debugger; the project contains the information needed to start the debug target (name of a routine, method, CSP page, Zen page, or client application). In addition, the project stores a list of breakpoints set in a prior debugging session for subsequent reuse.

You may attach and break into a running process without having a project open. In this case Studio does not remember breakpoint settings from previous sessions. See more about debugging in the chapter “Using the Studio Debugger.”

2.8.1 Debugging Object-Based Applications

At this time, Studio only allows source-level debugging of INT (ObjectScript routine) and BAS (Basic routine) files. To step through, or set breakpoints within classes or CSP pages, open the corresponding INT or BAS file and use the debugging commands in it.

To make sure that the generated source code for a class is available, check the `Keep Generated Source Code` option, on **Tools > Options** dialog, **Compiler > General Flags** tab.

2.9 Integration with Caché Security

Caché security features control both the use of Studio and the ability of Studio to connect to any Caché server. When you start Studio, it presents a login screen; to use Studio, you must log in as a user who holds the following privileges:

- **`%Development:Use`** - Use permission on the %Development resource grants access to various development-related resources.
- **`%Service_Object:Use`** - Use permission on the **`%Service_Object`** resource grants access to the %Service_Bindings service, which controls access to Studio.

Also, you can only connect to a namespace if you have Read or Write permission for its default database.

The way in which a user is granted these various privileges depends on the instance's security level, described in the list below. To change the Studio authentication settings, use the **Allowed Authentication Modes** check boxes on the **Edit Service** page for the **`%Service_Bindings`** service.

- For an instance with minimal security, all users, including UnknownUser, have all privileges and access to all namespaces. When presented with the Studio login screen, either leave the **Username** and **Password** fields blank or enter "_SYSTEM" and "SYS" as the username-password pair.
- For an instance with normal security, you must be explicitly granted the specified privileges. This is established by being assigned to a role or roles that holds these privileges.
- For an instance with locked-down security, the service that governs access to Studio (**`%Service_Bindings`**) is disabled by default. By default, no user has access to Studio.

Note: Studio access may also be affected by any changes to default settings that have occurred since installation.

2.10 Source Control Hooks

Studio includes a mechanism for implementing custom *hooks*—code that is executed on the Caché server whenever a document is loaded or saved. Typically these hooks are used to implement a connection to a source or revision control system.

To define source control hooks, create a subclass of the %Studio.SourceControl.Base class and implement the callback methods that you want. You can specify which Source Control class Studio should use by navigating to **System Administration** > **Configuration** > **Additional Settings** > **Source Control** on the Management Portal.

Refer to the %Studio.SourceControl.Base class and the "Using Studio Source Control Hooks" appendix for more details.

3

Building a Simple Application with Studio

This chapter contains a tutorial that illustrates some basic features of Studio. The tutorial shows you how to create a database application that is a phone book, containing names and phone numbers.

The phone book application consists of:

- A database in which you can store names and phone numbers.
- An interactive user interface on a Web page (an HTML form) that allows you to add new entries, search for entries, and edit entries in the phone book.

The tutorial demonstrates how to use Studio to:

1. Create a Caché Project to manage the source code for the application.
2. Define the application's database using a persistent database class.
3. Create a Web-based (HTML) user interface for the application using Caché Server Pages (CSP).
4. Create a Web-based (HTML) user interface for the application using InterSystems Zen.

3.1 Creating a Project

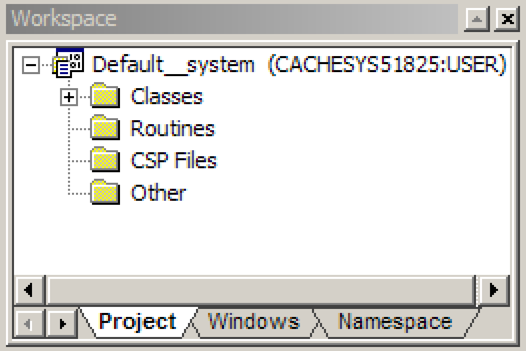
First, create a new project named `Phone Book` to manage the source files for the application, as follows:

1. Start Studio; right-click the Caché cube and select **Studio**.

   Studio connects to the local Caché server using the namespace used most recently by Studio (or displays the namespace dialog if this is the first time Studio is connecting).

   If Studio is not connected to the namespace in which you want to work, connect to a different namespace using the **File > Change Namespace**.

   By default, Studio displays the Workspace window and creates a new project called `Default_system`. The Studio Workspace window indicates the name of the current project as well as the server connection and namespace name. The Workspace window should be displayed by default; if you don't see it, display it using the **View > Workspace** or with the **Alt-3** keyboard accelerator.



2. To save your new project, select **File > Save Project As** and enter `Phone Book`.

You can save this project to the Caché server at any time using the **File > Save Project**.

3.2 Creating a Database

The Phone Book application is a database containing people's names and phone numbers, stored using a set of persistent objects. These objects are defined by a persistent class called Person. For sake of organization you can place this class in a package called `PhoneBook`.

The persistent class Person contains two properties (or fields): Name and PhoneNumber.

3.2.1 Defining a New Class

You can define this new Person class using Studio's New Class wizard by following these steps:

1. Start the New Class wizard by selecting **File > New > General tab**.
2. Double-click **Caché Class Definition**.
3. On the first page of the New Class wizard, enter a package name, `PhoneBook` and a class name, `Person`. Select **Next**.
4. Select `Persistent` from the list of available class types. Select **Finish**.

You see a Class Editor window containing the definition of your new class:

```
Class PhoneBook.Person Extends %Persistent
{
}
```

3.2.2 Adding Properties

Add the Name and PhoneNumber properties to the definition of the Person class with the New Property wizard as follows:

1. Select **Class > Add > Property** to start the New Property wizard.
2. Enter a name for the new property, Name, andselect **Finish**.

   Your class definition includes a property definition:

   ```
   Class PhoneBook.Person Extends %Persistent
   {
       Property Name As %String;
   }
   ```

3. Select **Class > Add > Property**. Enter PhoneNumber. Select **Finish**.

```
Class PhoneBook.Person Extends %Persistent
{
    Property Name As %String;
    Property PhoneNumber As %String;
}
```

You could have also added the PhoneNumber property by copying, pasting, and modifying the Name property directly into the Class Editor window. Items are indented to structure the code to improve readability. A plus/minus expansion box is provided to the left of each item so that you can collapse sections of the code that you are not currently looking at.

3.2.3 Saving and Compiling Your Class

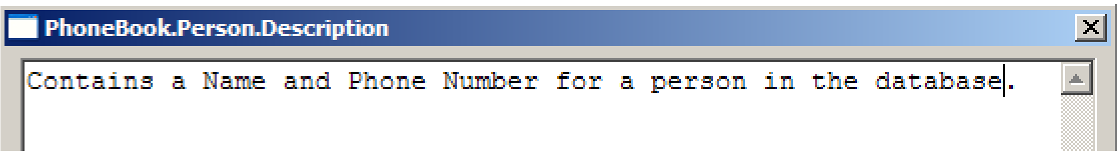
Save this class definition to the database and compile it into executable code with **Build > Compile** or select the **Compile** icon .

You now have a PhoneBook.Person class defined in your database.

3.2.4 Viewing Documentation for Your Class

View the automatically generated documentation for this class in the Caché online class reference with **View > Show Class Documentation**. To enter descriptions for your class and properties so that they appear in the online documentation, you can enter descriptions above class member declarations using /// (three slashes) or you can do the following:

1. Select the Class Inspector and view Class keywords (make the left column header of the inspector display the word `Class`)
2. In the Inspector, double-click the keyword `Description`.
3. This opens an editor in which you can enter a description for your class, such as the following:



```
Contains a Name and Phone Number for a person in the database.
```

4. Select **OK** when you are finished.
5. Save your class and view the documentation again.

3.3 Creating a Web User Interface using CSP

The user interface of the Phone Book application is a Web page containing an HTML form that lets you edit and view data, name and number, for a person in the database. You create this Web page using a CSP file that contains a form bound to the persistent PhoneBook.Person class.

3.3.1 Creating a CSP File

You can create a CSP file in Studio using the Web Form wizard as follows:

1. Create a new CSP file by selecting **File > New > CSP File tab > Caché Server Page**.

   A CSP Editor window is displayed containing source for the new CSP page entitled Unititled.csp.

2. Select **File > Save**.
3. In the `Save As` dialog, double-click /csp/user to open this directory. Enter Person.csp. Select **Save As**.
4. In the editor window, position the cursor in the <BODY> section of the CSP source file. Delete the words "My page body." Select **Tools > Templates > Templates**. Select **Web Form Wizard** from the list.
5. Select the PhoneBook.Person class and select **Next**.
6. Select the Name and PhoneNumber properties from the list of available properties. They should be displayed in the `Selected Properties` list.
7. Select **Finish**.

The Web Form Wizard places HTML source for a bound form in the CSP Editor window:

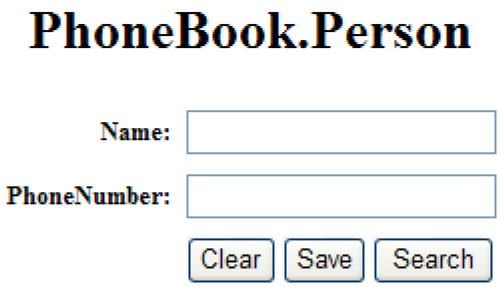
```
<html>
<head>

<!-- Put your page Title here -->
<title>        Cache Server Page </title>

</head>

<body>
<!-- Put your page code here -->

<head>
<title>Cache Server Page - PhoneBook.Person (USER)</title>
</head>
<h1 align='center'>PhoneBook.Person</h1>
<!-- This function is needed by the search button on the form -->
<script language='javascript'>
<!--
function update(id)
{
     #server(..formLoad(id))#;
     return true;
}

// -->
</script>

<!-- use CSP:OBJECT tag to create a reference to an instance of the class -->
<csp:object name='objForm' classname='PhoneBook.Person' OBJID='#(%request.Get("OBJID"))#'>

<!-- use csp:search tag to create a javascript function to invoke a search page -->
<csp:search name='form_search' classname='PhoneBook.Person' where='%Id()' options='popup,nopredicates'

onselect='update'>

<form name='form' cspbind='objForm' cspjs='All' onsubmit='return form_validate();'>
<center>
<table cellpadding='3'>
        <tr>
                <td><b><div align='right'>Name:</div></b></td>
                <td><input type='text' name='Name' cspbind='Name' size='50'></td>
        </tr>
        <tr>
                <td><b><div align='right'>PhoneNumber:</div></b></td>
                <td><input type='text' name='PhoneNumber' cspbind='PhoneNumber' size='50'></td>
        </tr>
        <tr>
                <td> </td>
                <td><input type='button' name='btnClear' value='Clear' onselect='form_new();'>
        <input type='button' name='btnSave' value='Save' onselect='form_save();'>
        <input type='button' name='btnSearch' value='Search' onselect='form_search();'></td>
        </tr>
</table>
</center>
</form>
</body>
</html>
```

3.3.2 Saving and Compiling Your CSP File

Save and compile the CSP file by selecting **Build > Compile** or **Ctrl-F7** or the **Compile** icon.

3.3.3 Viewing Your Web Page

View the Web page in a browser by selecting **View > Web Page** or the **Web Page** icon.



To learn more about using CSP to create Web pages, see the book *Using Caché Server Pages (CSP)*.

3.4 Creating a Web User Interface using Zen

Zen supports several approaches to creating Web-based forms like the CSP form **PhoneBook.Person** in the previous section. The same set of technologies provides the foundation for both CSP and Zen. Zen makes the development of fully-featured Web applications faster while building on the client/server communication features that CSP provides. The CSP/Zen relationship is explained further in the book *Using Zen*.

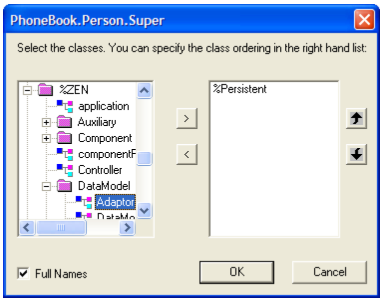
The tutorial in this section creates a Zen page, a Web-based user interface to the **Phone Book** project you began in this chapter. The general steps (details follow) are:

1. Make your class a Zen data adaptor
2. Create a Zen page
3. Add a Zen form
4. Use the form to add database entries
5. Add a Zen table to display database entries

3.4.1 Making Your Class a Data Adaptor

This step takes the PhoneBook.Person class that you developed for the project you began in this chapter, and converts it to work as a Zen data adaptor. This is the quickest way build a Zen user interface based on an existing class:

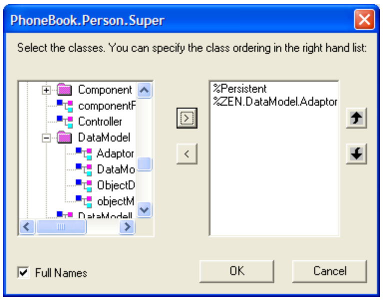
1. Open the PhoneBook.Person class.
2. Display the Studio Inspector window by selecting **View > Inspector**. If not already showing, in the left column header, select **Class** from the drop-down list. An alphabetical list of class keywords displays.
3. Find the **Super** keyword and select to highlight. The field currently holds the class name %Persistent and an ellipsis (...). Select the ellipsis. This opens a dialog in which you can choose superclasses in addition to %Persistent.



PhoneBook.Person.Super

Select the classes. You can specify the class ordering in the right hand list:

4. In the left-hand column of the dialog, navigate to the %ZEN.DataModel.Adaptor class and select to highlight. At the center of the dialog, select **>**, the right-angle bracket to place the %ZEN.DataModel.Adaptor class into the right-hand column underneath %Persistent.



PhoneBook.Person.Super

Select the classes. You can specify the class ordering in the right hand list:

5. Select **OK**.

6. Save and compile the PhoneBook.Person class.

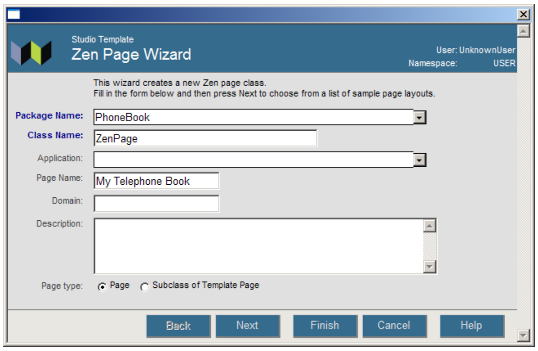
3.4.2 Creating a Zen Page

This step creates a ZenPage class that you can edit to create the user interface for your project:

1. Choose **File > New** or **Ctrl-N** or the **New** icon .

2. Select the **Zen** tab. Select **New Zen Page**. Select **OK**.

   . The Zen Page Wizard displays:



Studio Template

Zen Page Wizard

User: UnknownUser

Namespace: USER

This wizard creates a new Zen page class.
Fill in the form below and then press Next to choose from a list of sample page layouts.

Package Name: PhoneBook

Class Name: ZenPage

Application:

Page Name: My Telephone Book

Domain:

Description:

Page type: ● Page ○ Subclass of Template Page

3. Edit the dialog as follows:

   - Enter the **Package Name**: `PhoneBook`
   - Enter the **Class Name**: `ZenPage`
   - Enter the **Page Name**: `My Telephone Book`
   - Select **Next**.

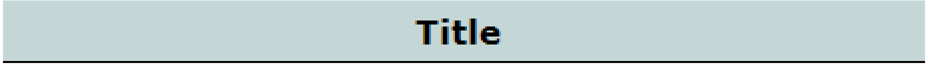
4. Select **Title Page** for your initial page layout. Select **Finish**.

   The Zen Page Wizard creates and displays a skeletal Zen page with template class parameters and the XML blocks XData Style and XData Contents. Notice the location of XData Contents in the class. You will edit this XML block to add items to your new Zen page:

   ```
   XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
   {
     <page xmlns="http://www.intersystems.com/zen" title="">
       <html id="title">Title</html>
       <vgroup width="100%">
         <!-- put page contents here -->
       </vgroup>
     </page>
   }
   ```

5. Save and compile with **Ctrl-F7**.

6. View the Web page by choosing **View > Web Page** or the **View Web Page** icon. The only item visible so far is the **Title** text displayed by the <html> element:

   Title

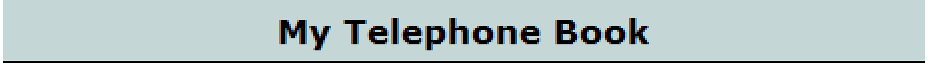
7. If you look at the XData Contents block, you see that the <html> element contains an attribute `id="title"`. `id="title"` refers to the style definition `#title` that appears in the XData Style block in the same ZenPage class. `#title` determines the background color, layout, and font choices that you see when you view the page. The default XData Style block for the **Title Page** layout looks like this:

   ```
   XData Style
   {
     <style type="text/css">
       /* style for title bar */
       #title {
         background: #C5D6D6;
         color: black;
         font-family: Verdana;
         font-size: 1.5em;
         font-weight: bold;
         padding: 5px;
         border-bottom: 1px solid black;
         text-align: center;
       }
     </style>
   }
   ```

8. Edit the text contents of the <html> element to provide a more meaningful title:

   ```
   <html id="title">My Telephone Book</html>
   ```

9. Save and compile the class, then view the Web page. It should look like this:

   My Telephone Book

3.4.3 Adding a Zen Form

Now that you have a ZenPage class to work with, you can edit the XML elements in its XData Contents block to add items to the display. In this exercise, you will begin by adding a form that allows you to add PhoneBook.Person objects to your database:

1. In Studio, open the ZenPage class.
2. Place <dataController> and <dynaForm> elements inside the main <vgroup> in XData Contents, exactly as shown in the following example:

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
  <page xmlns="http://www.intersystems.com/zen" title="">
    <html id="title">My Telephone Book</html>
    <vgroup width="100%">
      <dataController id="source" modelClass="PhoneBook.Person" modelId=""/>
      <dynaForm id="MyForm" controllerId="source" />
    </vgroup>
  </page>
}
```

   Place the cursor between <vgroup> and </vgroup> and begin typing. (Delete the **put page contents here** line.)

   After you type the character **<** Studio Assist displays a list of all elements. Typing **<d** brings you to the part of the list that includes <dataController> and <dynaForm>. Double-click on one of the choices to select it. Then type a space character and Studio Assist prompts you displays a list of attributes appropriate for that element. Continue in this manner until you have entered the entire line.

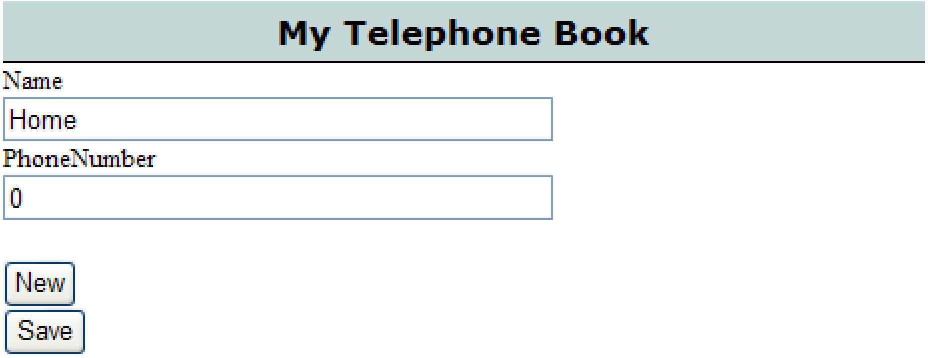
   Alternatively, if you are viewing this document online, you can cut and paste the <dataController> and <dynaForm> lines from the example above.

3. Provide two <button> elements in XData Contents, exactly as shown below:

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
  <page xmlns="http://www.intersystems.com/zen" title="">
    <html id="title">My Telephone Book</html>
    <vgroup width="100%">
      <dataController id="source" modelClass="PhoneBook.Person" modelId=""/>
      <dynaForm id="MyForm" controllerId="source" />
      <button caption="New" onselect="zenPage.newRecord();" />
      <button caption="Save" onselect="zenPage.saveRecord();" />
    </vgroup>
  </page>
}
```

   Type the new elements or, if you are viewing this document online, you can cut and paste the <button> lines from the example.

4. Save and compile with **Ctrl+F7**. View the Web page. It should look like this:


My Telephone Book

Name

Home

PhoneNumber

0

5. If you are curious about selecting the buttons **New** and **Save**, try it. An error message displays. Select **OK** to dismiss it.

   To understand why you saw an error message when you selected the buttons, look carefully at the values of the *onselect* attribute for each <button> element in XData Contents. Each *onselect* value is a JavaScript expression that executes automatically whenever the button is selected.

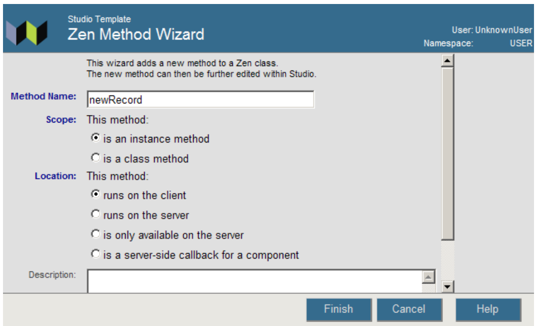
In these examples, the *onselect* expression invokes a method that runs in JavaScript on the client. The special variable `zenPage` indicates that the method is defined in the current ZenPage class. The methods themselves are called **newRecord** and **saveRecord**.

In order to make these *onselect* values work, you must create client-side JavaScript methods in the ZenPage class. This is quite simple to do using the Zen Method Wizard in Studio.

3.4.4 Adding Client-side Methods

In this step you add methods that create new objects and save them in response to button selects. These methods permit you to use your Zen form to populate your database with objects of the PhoneBook.Person class:

1. In Studio, open the ZenPage class.
2. Position the cursor below the closing curly bracket of the XData Contents block, but before the closing curly bracket for the ZenPage class.
3. Choose **Tools > Templates > Templates** or press **Ctrl-T** to display the Studio Template dialog. Choose **Zen Method Wizard**. Select **OK**. The following dialog displays:



This wizard adds a new method to a Zen class.
The new method can then be further edited within Studio.

Method Name: newRecord

Scope: This method:
● is an instance method
○ is a class method

Location: This method:
● runs on the client
○ runs on the server
○ is only available on the server
○ is a server-side callback for a component

Description:

Edit the dialog as follows. (Note that you will need to scroll down to the bottom of the dialog.)

- Enter the **Method Name** `newRecord`
- Choose **is an instance method**
- Choose **runs on the client**
- Provide a **Description** as shown.
- Clear the **Try/Catch** check box.
- Select **Finish**. Your new method appears in the page class as follows:

```
/// Create a new instance of the controller object.
ClientMethod newRecord() [Language = javascript]
{
  // TODO: implement
  alert('Client Method');
}
```

4. Change the code within curly brackets so the method now looks like this:

```
/// Create a new instance of the controller object.
ClientMethod newRecord() [Language = javascript]
{
  var controller = zenPage.getComponentById('source');
  controller.createNewObject();
}
```

5. Repeat the above steps to add the **saveRecord** method. When using the Zen Method Wizard, enter values in the dialog as follows:

- Enter the **Method Name** `saveRecord`
- Choose **is an instance method**
- Choose **runs on the client**
- Provide a **Description** of `Create a new instance of the controller object`.
- Clear the **Try/Catch** check box.
- Select **Finish**. Your new method appears in the page class.

Edit the new method so that it looks like this:

```
/// Save the current instance of the controller object.
ClientMethod saveRecord() [Language = javascript]
{
  var form = zenPage.getComponentById('MyForm');
  form.save();
}
```

6. Save and compile the class, then view the Web page.
7. Select the **New** button on the page. The Name and PhoneNumber fields become empty so that you can enter new information for the next entry. After you have typed in each field, select the **Save** button. The new entry is saved in the database.
8. Use **New** and **Save** repeatedly to add more entries.

3.4.5 Viewing the Database in a Table

Now that you have something in your database, you would like to be able to see it. In this step you will add a Zen table that displays the saved objects from your database. You will then modify the **saveRecord** method so that it automatically updates this table each time you select the **Save** button in the user interface:

1. In Studio, open the ZenPage class.
2. Provide one <tablePane> element inside the main <vgroup> in XData Contents, exactly as shown below. You can type the new element, or if you are viewing this document online, for convenience you may cut and paste the <tablePane> lines from this example:

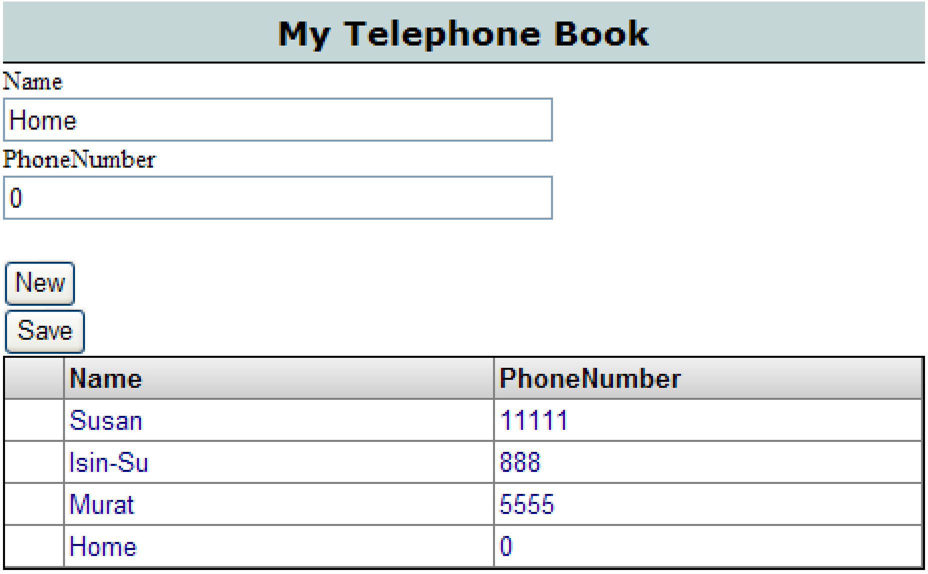
```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
  <page xmlns="http://www.intersystems.com/zen" title="">
    <html id="title">My Telephone Book</html>
    <vgroup width="100%">
      <dataController id="source" modelClass="PhoneBook.Person" modelId=""/>
      <dynaForm id="MyForm" controllerId="source" />
      <button caption="New" onselect="zenPage.newRecord();" />
      <button caption="Save" onselect="zenPage.saveRecord();" />
      <tablePane id="people"
                 sql="SELECT Name,PhoneNumber FROM PhoneBook.Person" />
    </vgroup>
  </page>
}
```

   The <tablePane> *sql* attribute provides an SQL statement. SELECT lists the two properties from your PhoneBook.Person class, and FROM provides the full package and class name. This SQL query provides the data for the <tablePane>.

3. Save and compile the class, then view the Web page.
4. Use **New** and **Save** to add more entries to the database.
5. Select the browser refresh button to view the updated table. The new entries are visible.
6. Remove the need for the user to refresh after each new entry, by refreshing the table automatically after each save. To accomplish this, add two lines to the **saveRecord** method, so that it looks like this:

```
/// Save the current instance of the controller object.
Method saveRecord() [Language = javascript]
{
  var form = zenPage.getComponentById('MyForm');
  form.save();
  var table = zenPage.getComponentById('people');
  table.executeQuery();
}
```

7. Save and compile the class, then view the Web page. It should look like this:



My Telephone Book

Name: Home

PhoneNumber: 0

	Name	PhoneNumber
	Susan	11111
	Isin-Su	888
	Murat	5555
	Home	0

8. Use **New** and **Save** to add more entries to the database. Each time you select **Save**, the **saveRecord** method updates the table so that the newest entry becomes visible.

To learn more about Zen, see the book *Using Zen*.

4

Creating Class Definitions

The Studio lets you create and edit class definitions. A class definition specifies the contents of a particular class including its members (such as methods and properties) and characteristics (such as superclasses).

With Studio you can work with class definitions with several tools:

- Wizards to quickly create classes and class members
- Class Inspector to view and edit class characteristics in a table
- Class Editor to directly edit the class definition. The Class Editor is a full-featured text editor that provides syntax coloring, syntax checking, and code completion drop-down menus of available options.

You can use all of these techniques interchangeably; Studio automatically ensures that all of these representations are synchronized.

This chapter discusses general aspects of creating class definitions. Most of the following chapters in this book describe how to create class members, such as properties, methods, parameters, and so forth.

4.1 Creating New Class Definitions

You can create a new class definition in Studio by using the New Class wizard.

Note: You must have an open project before you can work with class definitions in the Studio. When working with class definitions, Studio performs numerous interactions with the Caché server (such as for providing lists of classes, class compiling, etc.). Internally, Studio uses projects to manage the details of this server interaction.

4.1.1 New Class Wizard

To open the New Class wizard, select **File > New** and select `New Class Definition`.

The New Class wizard prompts you for information. Select **Finish** at any time (in this case, default values are provided for any information you have not specified).

4.1.1.1 Name and Description Page

The New Class wizard prompts you for the following information (with the exception of class and package name, you can later modify any of these values):

Package Name

Package to which the new class belongs. You can select an existing package name or enter a new name. If you enter a new name, the new package is automatically created when you save your class definition. The only punctuation marks that property names can contain are a dot (.) and a leading percent sign (%).

For more information on packages, see the chapter "Packages" in *Using Caché Objects*.

Class Name

Name of your new class. This must be a valid class name and must not conflict with the name of a previously defined class. Note that you cannot change this class name later.

For specifics of class naming conventions in Caché, see the section "Naming Conventions" in *Using Caché Objects*.

Description

(optional) Description of the new class. This description is used when the class' documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see "Using HTML Markup in Class Documentation" in the chapter "Defining and Compiling Classes" in *Using Caché Objects*.

4.1.1.2 Class Type Page

The New Class wizard asks you what type of class you would like to create. You can either extend (inherit from) a previously defined class or create a new class by selecting one of the following options:

Persistent

Create a definition for a persistent class. Persistent objects can be stored in the database.

Serial

Create a definition for a serial class. Serial objects can be embedded in persistent objects to create complex data types such as addresses.

Registered

Create a definition for a registered class. Registered objects are not stored in the database.

Abstract

Create a definition for an abstract class with no superclass.

Datatype

Create a definition for a data type class. A data type class is used to create user-defined data types.

CSP (used to process HTTP events)

Create a definition for a %CSP.Page class. A CSP class is used to create a CSP event handling class. This is a programmatic way to create CSP Pages or to respond to HTTP events (for example, to create an XML server).

Extends

Extend an existing class: check `Extends` and enter (or choose from a list) the name of an existing superclass.

4.1.1.3 Data Type Class Characteristics Page

If you are creating a new data type class, the New Class wizard prompts for certain items particular to data type classes. These include:

Client Data Type

The data type used by clients, such as Active X, to represent this data enter a client application.

ODBC Data Type

The data type used by ODBC or JDBC to represent this data type. Choose a type that corresponds to how you want this data type to appear to ODBC/JDBC based applications.

SQL Category

The SQL Category used by the Caché SQL Engine when it performs logical operations on this data type.

4.1.1.4 Persistent, Serial, Registered Class Characteristics Page

If you are creating a new persistent, serial class, or registered, the New Class wizard prompts for certain items particular to persistent or serial classes. These include:

Owner

(optional) For a persistent class, enter the SQL username to be the owner of the new class. This username controls privileges when this class is used via SQL. If this field is left blank, then the default owner, `_system`, is used.

SQL Table Name

(optional) For a persistent class, enter a name to be used for the SQL table that corresponds to this class. If this field is left blank, then the SQL table name is identical to the class name. If the class name is not a valid SQL identifier, you must enter an SQL table name here.

XML Enabled

(optional) If selected, the class is XML-enabled; that is, it has the ability to project itself as an XML document. It can also be used in Web Service methods. This is equivalent to adding the %XML.Adaptor class to the class' superclass list.

For more information see *Using XML with Caché* as well as *Creating Web Services and Web Clients in Caché*.

Zen DataModel

(optional) If checked, the class includes `extends %ZEN.DataModel.Adaptor`.

Data Population

(optional) If you select this option, your new class supports automatic data population. This is equivalent to adding the %Library.Populate class to the class' superclass list.

Automatic data population allows you to easily create random data with which you can test the operation of your class. To populate a class, compile it and then execute the class' **Populate** method (inherited from the %Library.Populate class). For example, using the Terminal:

```
 Do ##class(MyApp.Person).Populate(100)
```

For more information see the chapter "Caché Data Population Utility" in *Using Caché Objects*.

MultiValue Enabled

The created class inherits from %MV.Adaptor, such that the class uses MV storage and the data appears as a “file” to MultiValue programs and queries.

4.1.1.5 CSP Class Characteristics Page

If you are creating a new CSP class, the New Class wizard prompts for the following value:

Content Type

Specifies what the content type served by this CSP class is. The available options are `HTML` or `XML.` This option is used to set the value of the *CONTENTTYPE* parameter of the new class to `text/html` or `text/xml` respectively. You can later change this to whatever value you want.

4.1.2 Results of Running the New Class Wizard

After running the New Class wizard, Studio displays a new Class Editor window. The Class Editor window contains your new class definition. For example:

```
/// This is a Person class
class MyApp.Person extends %Persistent
{
}
```

You can save this class definition in the Caché database, add class members such as properties or methods, or edit the class definition using the Class Inspector.

Note: By default, the New Class wizard creates classes that use ObjectScript for method code. You can change this default value to Basic with **Tools > Options** dialog, **Environment, General tab**.

4.2 Opening Class Definitions

You can open a previously saved class definition and display it in a Class Editor window by selecting the class in the Project tab of the Workspace window and double-clicking it with the mouse.

If the class definition you want to open is not part of the current project, first add it to the current project using **Project > Add Class**.

If the class definition you want to open is currently being edited by someone else, you are asked if you want to open the class definition for read-only access.

4.3 Editing Class Definitions

You may modify any of the characteristics of a newly created or previously existing class definition (with the exception of the class or package name). You can do this in two ways:

- Using the Class Inspector to change the value of a class or class member keyword.
- Changing a value in the class definition using the Class Editor.

For the class keywords and other details on definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*

4.4 Saving and Deleting Class Definitions

If you have modified a class definition, save it to the Caché database in either of the following ways:

- Use **File > Save** to save the contents of the current window.
- Use **Save Project** to save all modified class definitions in the current project.

To delete a class definition, in the Workspace window, select a class and select **Edit > Delete class** *classname*. The class and all of its generated files are deleted.

4.5 Compiling Class Definitions

You can compile class definitions from Studio in these ways:

- Using **Build > Compile** or the **Compile** icon, . This saves all modified class definitions and compiles the current class definition (the one displayed in the active editor window).
- Using **Build > Rebuild All** or the **Rebuild All** icon, . This saves all open, modified class definitions and compiles all classes in the current project.

Note: You can control how classes are compiled using options on **Tools > Options** dialog, **Compiler** tab).

4.5.1 Incremental Compilation

The Studio can do incremental compilation of classes. The feature is enabled with the **Skip Related Up-to-date Classes** option. To find this option, open the **Tools > Options** dialog, **Compiler, General Flags** tab.

When enabled, if changes have been made to source code in one or more methods, only those methods are compiled with **Build > Compile**. (Use **Build > Rebuild All** to override.) Any changes to the class interface (properties, method signatures, etc.) or storage definition cause a full compilation.

Incremental compilation is typically much faster than a full compilation and speeds up the process of making incremental changes to methods (application logic) during development.

Incremental compilation works as follows:

1. The Class Compiler finds all methods whose implementation has changed, places their runtime code into a new routine, such as MyApp.MyClass.5.INT, and compiles this routine.
2. The Class Compiler then modifies the runtime class descriptor for the class to use the new implementations of the compiled methods. When an application invokes one of these methods, the new code is dispatched to and executed.
3. The rest of the class definition (compiled meta-information, storage information for persistent classes, runtime SQL information is left unchanged. Note that the previous implementation of the modified methods remains in the runtime code but is not executed.

When a full (non-incremental) compilation is performed, all of the extra routines containing incrementally compiled methods are removed. Perform a full compilation on all classes before deploying an application to avoid having extra routines.

4.6 Renaming Class Definitions

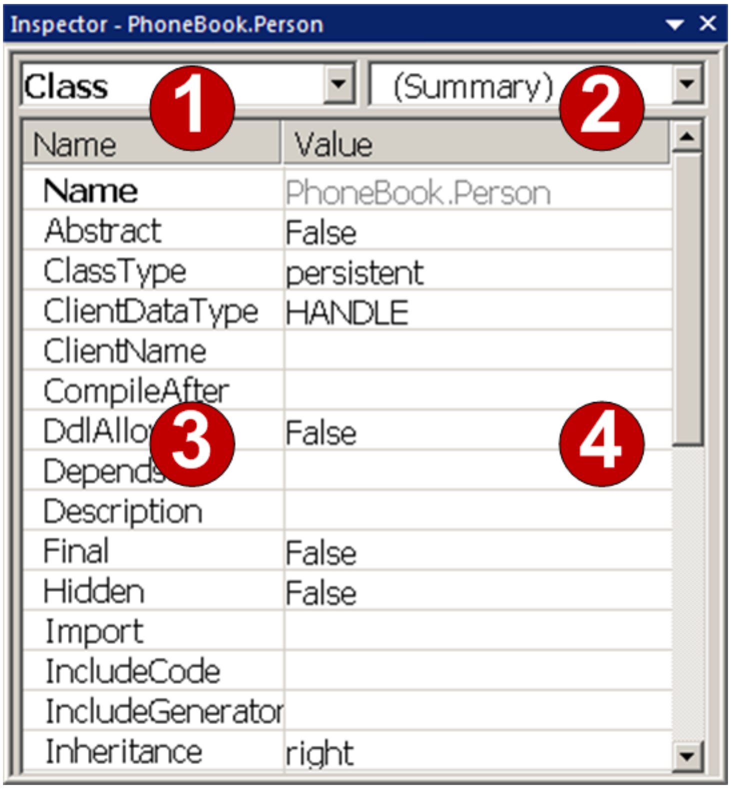
Once you have created a class definition you cannot change its name. You can perform the equivalent of this operation by creating a copy of the class with a new name as follows:

1. Select **Tools > Copy Class**.
2. Select the class you want to rename in the **From** field.
3. Enter the new class name in the **To** field.
4. Select any of the three options: `Add new class to project`, `Replace instances of the class name`, or `Copy Storage Definition`.
5. Select **OK**.
6. You have a new Class Editor window containing a copy of the original class definition. Using the text editor, you can make any additional changes you desire. You can save this new class definition when you like.
7. You can, if you want, delete the old class definition.

4.7 Class Inspector

The Class Inspector displays the current class definition in an editable table. The main components of the Class Inspector are described below:

Figure 4–1: Class Inspector



Name	Value
Name	PhoneBook.Person
Abstract	False
ClassType	persistent
ClientDataType	HANDLE
ClientName	
CompileAfter	
DdlAllo	False
Depends	
Description	
Final	False
Hidden	False
Import	
IncludeCode	
IncludeGenerator	
Inheritance	right

1. Member Selector: Controls which set of keywords are displayed. You can choose to view either the Class-wide keywords or the keywords for a specific class member (such as properties or methods).
2. Item Selector: Controls which specific class member is displayed (such as a particular property). The contents of the list depend on the value of the Member Selector. Selecting `(Summary)` displays a list of all the members of the type specified by the Member Selector.
3. Keywords: Lists the keywords for the current class or class member selected by the Member and Item Selectors. Highlighting a keyword displays its description and allows editing select **Edit** at the right of the value or directly edit the value). Keywords whose value was set explicitly (not inherited or set by default) are shown in bold.
4. Values: Lists the values of keywords displayed in the keyword list. Values modified since the last time the class definition has been saved are displayed in blue.

4.7.1 Activating the Class Inspector

The Class Inspector displays current information when it is activated (it is gray when inactive). To activate the Class Inspector:

1. Make sure that the current editor window contains a class definition (the Class Inspector does not work with Routines or CSP files).
2. Select the Class Inspector

When the Class Inspector is activated its background turns white and its contents are updated to reflect the current class definition. If you modify any keyword values using the inspector, the corresponding Class Editor window becomes inactive (turn gray). When you are finished with the inspector, select the original Class Editor window. It becomes active and display the result of the modifications you made using the Class Inspector.

If you right-click in the Class Inspector, it displays a popup menu allowing you to perform operations such as adding new class members.

4.8 Class Browser

Studio includes a class browsing utility that lets you view all available classes arranged by class hierarchy. Within each class you can view class members such as properties and methods, including those inherited from superclasses. The Class Browser displays class members in a table. By select a column title, you can sort the class members by that column.

1. Open the Class Browser with **Tools > Class Browser**.
2. Right-click an item in the Class Browser and select whether to add it to the project, open it in the Class Editor, or view documentation.

4.9 Superclass Browser and Derived Class Browser

Studio includes two additional browsers, one for listing all superclasses and one for listing derived classes from the current class definition.

4.9.1 Superclass Browser

Open the Superclass Browser using **Class > Superclasses** to display an alphabetical list of all superclasses of the current class.

Select a class and then select a button to either add it to the current project, open it in the Class Editor, or view documentation.

4.9.2 Derived Class Browser

Open the Derived Class Browser using **Class > Derived Classes** to display a list, in alphabetical order, of all the classes derived from the current class definition.

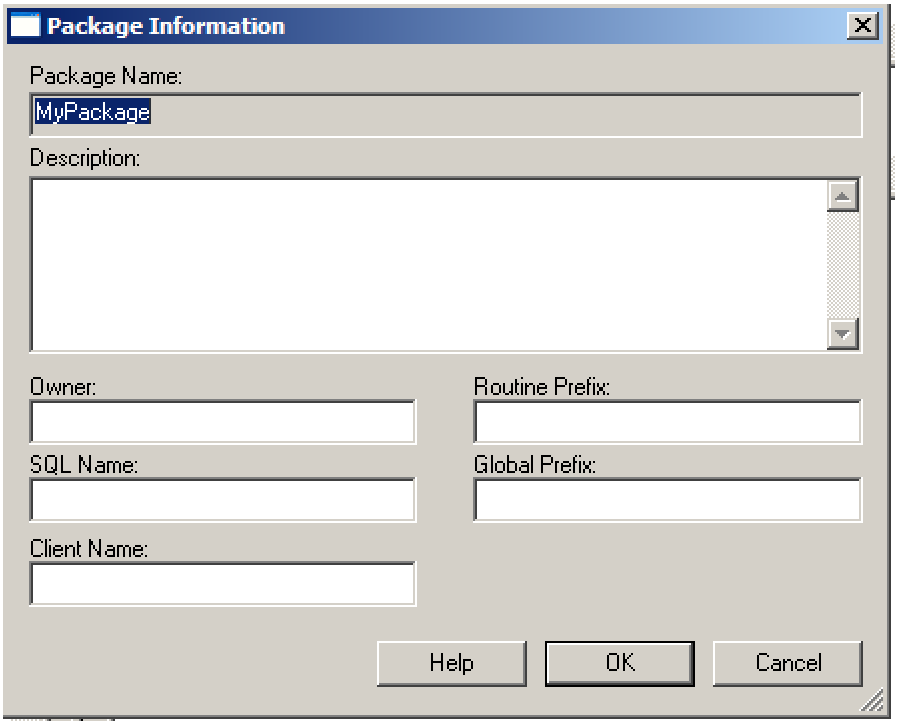
Select a class and then select a button to either add it to the current project, open it in the Class Editor, or view documentation.

4.10 Package Information

Within Studio, you can view and edit information about a specific class package using the Package Settings dialog.

To open Package Information, in the Workspace window, in the `Project` tab, right-click the package name and select **Package Information**.

Figure 4–2: Package Settings dialog



Package Information

Package Name: MyPackage

The Package Information window displays the following information:

Package Name	Name of the package.
Description	Description of the package
Owner	SQL Owner name of this package. This is used to provide Schema-wide privileges to the SQL representation of the package.
SQL Name	Name of the SQL Schema used to represent the package relationally.
Client Name	Package name used for the generated projection of this package's classes. For example, if this package contains a class named `bank.account`, and you give it a client package name of `com.mycompany.bank`. when the class is compiled, a Java projection of this class is put into `com.mycompany.bank.account`.
Routine Prefix	String that is used as a prefix for the routines generated from classes in this package
Global Prefix	String that is used as a prefix for the default global names used by persistent classes in this package

For more information on class packages see the chapter “Packages” in *Using Caché Objects*.

5

Adding Properties to a Class

This chapter describes how to add properties in a class definition.

The data, or state, of an object is stored in it properties. Every class definition may contain zero or more property definitions.

You can add a new property to a class in two ways:

- Adding the property to the class definition in the Class Editor.
- Using the New Property wizard

To add a property using the Class Editor, position the cursor on a blank line in the Class Editor and enter a property declaration:

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Title As %String;
}
```

Alternatively, copy an existing property declaration, paste it into a new location, and edit it.

For details on property definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*.

5.1 New Property Wizard

To open the New Property wizard, select **Class > Add > Property** . Alternatively, right-click the Class Inspector and select **Add > Property** or, if only properties are displayed (`Property` heads the left column), right-click and select **New Property** or select the **New Property** icon, from the toolbar.

The New Property wizard prompts you for information. Select **Finish** at any time; default values are provided for any information you have not specified.

5.1.1 Name and Description Page

The New Property wizard prompts you for the following information (you can later modify any of these values):

Property Name

(required) Name of the new property. This name must be a valid property name and must not conflict with the name of a previously defined property. The only punctuation marks that property names can contain are a dot (.) and a leading percent sign (%).

For a general discussion on names see the chapter “Caché Classes ” in *Using Caché Objects*.

Description

(optional) Description of the new property. This description is used when the class' documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see “Using HTML Markup in Class Documentation” in the chapter “Defining and Compiling Classes” in *Using Caché Objects*.

5.1.2 Property Type Page

The New Property wizard asks you to select the property type: single-valued, a collection, a stream, or a relationship. You can further refine each of these choices by specifying additional characteristics, such as data type.

Single Valued

A single-valued property is just that; it contains a single value. A single-valued property has a type associated with it. This type is the name of a Caché class. If the class used as a type is a data type class, then the property is a simple literal property; if it is a persistent class, then the property is a reference to an instance of that class; if it is a serial class, then the property represents an embedded object.

You can enter a class name directly or choose from a list of available classes, including streams, using the **Browse** button.

For a description of the basic data type classes provided with Caché, see the chapter “Data Types” in *Using Caché Objects*.

Collection

A collection property contains multiple values. There are two collection types, List (a simple, ordered list) and Array (a simple dictionary with elements associated with key values). As with a single-valued property, a collection property also has a data type. In this case, the data type specifies the type of the elements contained in the collection.

Relationship

A relationship property defines an association between two objects. For more details on relationships, see the chapter “Adding Relationships to a Class” in this book.

5.1.3 Property Characteristics Page

If you are adding a new property to a class definition for a persistent or serial object, the New Property wizard asks for additional characteristics. These include:

Required

(optional) This is only relevant for persistent or serial classes. Specifies that this property is required (NOT NULL in SQL terminology). For persistent or serial objects a required property must be given a value or any attempt to save the object fails.

Indexed

(optional) This is only relevant for persistent classes. Specifies that an index should be created based on this property. This is equivalent to creating an index based on this field.

Unique

(optional) This is only relevant for persistent classes. Specifies that the value of this property must be unique in the extent (set of all) objects of this class. This is equivalent to creating a unique index based on this field.

Calculated

(optional) A calculated property has no in-memory storage allocated for it when an object instance is created. Instead, you must provide accessor (Get or Set) methods for the property. If you choose this option, the New Property wizard can generate an empty Get accessor method for you.

SQL Field Name

(optional) In the case of a persistent class, this is the name that should be used for the SQL field that corresponds to this property. By default (when this field is blank) the SQL field name is identical to the property name. Provide an SQL field name if you want to use a different field name or if the property name is not a valid SQL identifier.

5.1.4 Data Type Parameters Page

Every property has a list of parameter values which is determined by the type of the property. The values of these parameters control aspects of the property's behavior. Using the table displayed on the Parameters page of the New Property wizard you can specify the value of particular parameters.

5.1.5 Property Accessors Page

You can override the Set method (used to set the value of a property) and Get method (used to retrieve the value of a property) for a property by selecting the corresponding override check box. Choosing one of these options creates an empty Set or Get method which you have to fill in later.

5.1.6 Results of Running the New Property Wizard

After running the New Property wizard the Class Editor window is updated to include the new property definition. For example:

```
/// This is a Person class
Class MyApp.Person extends %Persistent
{
Property Name As %String;
}
```

If you want to make further modifications to this property you can do this using either the Class Editor or the Class Inspector.

6

Adding Methods to a Class

This chapter discusses how to add and edit method definitions in a class definition.

The operations that are associated with an object and can be performed by it are referred to as methods. Every class definition may contain zero or more methods.

You can add a new method to a class definition in two ways:

- Adding a method to the class definition using the Class Editor.
- Using the New Method wizard

To add a method using the Class Editor, position the cursor on a blank line in the Class Editor and enter a method declaration:

```
Class MyApp.Person Extends %Persistent
{
Method NewMethod() As %String
{
    Quit ""
}
}
```

Alternatively, you can do this by copying and pasting an existing method declaration and then editing it.

For details on method definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*.

6.1 New Method Wizard

You can invoke the New Method wizard using the **Class > Add > Method**. Alternatively, right-click the Class Inspector and select **Add Method** or select the **New Method** icon from the toolbar.

The New Method wizard prompts you for information. Select **Finish** at any time (default values are provided for any information you have not specified).

6.1.1 Name and Description Page

The New Method wizard prompts you for the following information (you can later modify any of these values):

Method Name

(required) Name of the new method. This name must be a valid method name and must not conflict with the name of a previously defined method.

For a general discussion on names see the chapter “Caché Classes ” in *Using Caché Objects*.

Description

(optional) Description of the new method. This description is used when the class' documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see “Using HTML Markup in Class Documentation” in the chapter “Defining and Compiling Classes” in *Using Caché Objects*.

6.1.2 Method Signature Page

Every method has a signature that indicates its return type (if any) as well as its formal argument list (if any). For method signature you may specify the following:

Return Type

(optional) Indicates the type of the value returned by this method. This type is the name of a Caché class. You can type this name in directly or choose from a list of available classes using the **Browse** button.

For example, a method that returns a true (1) or false (0) value would have a return type of %Boolean. Leave this field empty if your new method has no return value.

For a description of the basic data type classes provided with Caché see the chapter “Data Types” in *Using Caché Objects*.

Arguments

(optional) Indicates the names, types, default values, and how data is passed (by reference or by value) for any formal arguments. The arguments are displayed in order in a table. You can add a new item to the argument list using **Add** located on the side of the table. This displays a popup dialog allowing you to specify the name of the argument, its type, its optional default value, and whether it is passed by value or by reference. Using the other buttons, you can remove and rearrange the order of items in the list.

6.1.3 Method Characteristics Page

You may specify additional characteristics for your method. These include:

Private

(optional) Indicates whether this method is public or private. Private methods may only be invoked from other methods of the same class.

Final

(optional) Indicates whether this method is final. Final methods cannot be overridden by subclasses.

Class Method

(optional) Indicates that the new method is a class method (as opposed to an instance method). Class methods may be invoked without having an object instance.

SQL Stored Procedure

(optional) Indicates that this method is accessible to an ODBC or JDBC client as a stored procedure. Only class methods may be projected as SQL Stored Procedures.

Language

Indicates the language used to create the method.

6.1.4 Implementation Page

If you want, you may enter the implementation (code) for the new method by typing lines of source code into the text editor window. You can also enter this source code after running the wizard using the Class Editor.

6.1.5 Results of Running the New Method Wizard

After running the New Method wizard the Class Editor window is updated to include the new method definition. You can edit it using either the Class Editor or the Class Inspector. For example:

```
/// This is a Person class
class MyApp.Person extends %Persistent
{
Method Print() As %Boolean
{
    Write "Hello"
    Quit 1
}
}
```

6.2 Overriding a Method

Note: The Refactor submenu is available only when Studio is connected to a Windows server. The Override menu item is available in other platforms.

One of the powerful features of object-based development is that classes inherit methods from their superclasses. In some cases, you may want to override, that is, provide a new implementation for, a method inherited from a superclass.

Class > Refactor > Override simplifies the process of overriding a class item, such as a specific method by displaying a list of all the methods defined by superclasses that can be overridden by the current class.

For example, in a persistent class, you may want to override the default implementation of the **%OnValidateObject** method provided by the %RegisteredObject class in order to specify custom validation that occurs when an instance of your class is saved.

To do this, follow these steps:

1. Open (or create) a persistent class definition in Studio.
2. Select **Class > Refactor > Override** and select the Methods tab. This displays a dialog window containing a list of methods which you can override.
3. Select **%OnValidateObject** from the list and select **OK** button.

Your class definition now includes a definition for an **%OnValidateObject** method:

```
class MyApp.Person extends %Persistent
{
// ...
Method %OnValidateObject() As %Status
{
}
}
```

At this point, you can use the Class Editor to add code to the body of the method.

7

Adding Class Parameters to a Class

A class parameter defines a constant value for all objects of a given class. When you create a class definition (or at any point before compilation), you can set the values for its class parameters. By default, the value of each parameter is the null string; to set a parameter's value, you must explicitly provide a value for it. At compile-time, the value of the parameter is established for all instances of a class. This value cannot be altered at runtime.

You can add a class parameter to a class definition in two ways:

- Adding a class parameter to the class definition using the Class Editor.
- Using the New Class Parameter wizard.

To add a method using the Class Editor, position the cursor on a blank line in the Class Editor and enter a class parameter:

```
Parameter P1 = "x";
```

7.1 New Class Parameter Wizard

You can use the New Class Parameter Wizard to create a new class parameter. You can open the New Class Parameter wizard using the **Class > Add > Class Parameter** . Alternatively, right-click the Class Inspector and select **Add New Class Parameter** or select the **New Class Parameter** icon on the toolbar.

The New Class Parameter wizard prompts you for information. Select **Finish** at any time (default values are provided for any information you have not specified).

See for more information about class parameters see the chapter “Caché Classes” in *Using Caché Objects*.

The New Class Parameter wizard prompts you for the following information (you can later modify any of these values):

Name

(required) Name of the class parameter. This name must be a valid parameter name and must not conflict with the name of a previously defined parameter.

For a general discussion on names see the chapter “Caché Classes” in *Using Caché Objects*.

Description

(optional) Description of the new class parameter. This description is used when the class' documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see "Using HTML Markup in Class Documentation" in the chapter "Defining and Compiling Classes" in *Using Caché Objects*.

Default

Default value for the class parameter. This will be the default value for this parameter for all instances of this class.

8

Adding Relationships to a Class

A relationship is a special type of property that defines how two or more object instances are associated with each other. For example, a Company class that represents a company could have a one-to-many relationship with an Employee class that represents an employee (that is, each company may have one or more employees while each employee is associated with a specific company).

A relationship differs from a property in that every relationship is two-sided: for every relationship definition there is a corresponding inverse relationship that defines the other side.

For more information on relationships, see the chapter "Relationships" in *Using Caché Objects*.

You can add a new relationship to a class definition in two ways:

- Adding a relationship to the class definition using the Class Editor.
- Using the New Property wizard

To add a relationship using the Class Editor, position the cursor on a blank line in the Class Editor and enter a relationship declaration:

```
Class MyApp.Company Extends %Persistent
{
Relationship TheEmployees As Employee [cardinality=many, inverse=TheCompany];
}
```

A relationship definition must specify values for both the cardinality and inverse keywords.

As this relationship has two sides, also enter the inverse relationship in the class definition of Employee:

```
Class MyApp.Employee Extends %Persistent
{
Relationship TheCompany As Company [cardinality=one, inverse=TheEmployees];
}
```

If the two sides of the relationship do not correctly correspond to each other, there are errors when you try to compile.

8.1 New Property Wizard to Create a Relationship Property

You can use the New Property Wizard to create a new relationship property. To invoke this wizard, use **Class > Add > Property**. Alternatively, right-click the Class Inspector and select **Add Property** or select **New Property** on the toolbar.

The New Property wizard prompts you for information. The procedure is identical to creating a new, non-relationship property except that you specify `Relationship` on the property type.

8.1.1 Name and Description Page

The New Property wizard prompts you for the following information (you can later modify any of these values):

Property Name

(required) Name of the relationship. This name must be a valid relationship (property) name and must not conflict with the name of a previously defined relationship or property.

For a general discussion on names see the chapter “Caché Classes” in *Using Caché Objects*.

Description

(optional) Description of the new relationship. This description is used when the class' documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see “Using HTML Markup in Class Documentation” in the chapter “Defining and Compiling Classes” in *Using Caché Objects*.

8.1.2 Property Type Page

The New Property wizard asks you to choose from a variety of property types. Choose `Relationship` and enter the name of the class on the inverse side of the relationship.

8.1.3 Relationship Characteristics Page

The New Property wizard asks for additional relationship characteristics. These include:

`Cardinality`

One: one other object

This relationship property refers to a single instance of the related object. The resulting property acts like a simple reference field.

Many: many other objects

This relationship property refers to a one or more instances of the related object. The resulting property acts like a collection of objects.

Parent: this object's parent

Identical to cardinality of `one` except that this is a dependent relationship and this property refers to the parent of this object. Note: When you create a parent-child relationship, you are not given the option to create an index because children aren't stored independently but within the parent; you can see that by looking at the global structure. The children are indexed automatically by creating an extra subscript.

Children: the object's children

Identical to cardinality of `many` except that this is a dependent relationship and this property refers to a collection of child objects.

`Inverse:`

This relationship property references objects of the following type

Select a class from the Browse button or enter a new class name for the inverse side of the relationship.

The name of the corresponding property in the referenced class

Select a property from the class or enter a new property name for the inverse side of the relationship.

8.1.4 Additional Changes

Select any of the additional changes that you would like to implement:

Create a new class “<inverse class>”

This field is active if you entered a class name that does not exist for the inverse relationship on the last screenSelect this to create the class in this package. The new class is added to the package. You must compile the new class before you can compile the class that contains the relationship.

Create a new property “Parent” in class “<inverse class>”

This field is active if you entered a property that does not exist for the inverse relationship on the last screenSelect this to create the property in the package and class named in the last screen. The new property is added to the class. You must compile the class with this new property before you can compile the class that contains the relationship.

Modify property “Parent” of class “<inverse class>”

This property is active if you entered a property that already exists for the inverse relationship on the last screenSelect this to modify that property to be a relationship with this property.

Define an index for this relationship.

Check to define an index for this property. This is applicable only for One-To-Many relationships; it is disabled for Parent-Child relationships

8.1.5 Results of Creating a New Relationship with the New Property Wizard

After creating a new relationship with the New Property wizard, the Class Editor window is updated to include the new relationship definition. For example:

```
/// This is an Employee class
class MyApp.Employee extends %Persistent
{

/// We have a one-to-many relationship with Company
Relationship Company As Company [cardinality=one, inverse=Employees];
}
```

If you want to make further modifications to this relationship, you can do this using either the Class Editor or the Class Inspector.

Additionally you can use the Modify Relationship wizard, which has the advantage of automatically determining the changes required to the inverse of a relationship.

You can invoke the Modify Relationship wizard by following these steps:

1. Display the list of properties in the Class Inspector
2. Right-click the desired relationship in the list of properties and select **Add/Modify Relationship** command from the pop-up menu.

9

Adding Queries to a Class

Caché class definitions may contain query definitions. Each query defines an instance of query interface: a set of methods that can be called from a Caché %ResultSet object allowing you to iterate over a set of data.

You can use queries in the following ways:

- Via the %ResultSet class in server-based methods.
- Via the Java ResultSet class that is included with the Caché Java binding.
- Via the ActiveX ResultSet class that is included with the Caché ActiveX binding.
- As ODBC or JDBC stored procedures (if you specify that the query should be projected as an SQL stored procedure).

There are two kinds of queries:

- Those based on SQL statements (using the %SQLQuery class) and
- Those based on user-written code (using the %Query class).

The %SQLQuery class automatically provides an implementation of the query interface. The %Query class does not; the query definition merely provides a query interface; you must write methods to implement the actual query functionality.

You can add a new query to a class definition in two ways:

- Edit the class definition using the Class Editor.
- Using the New Query wizard

To add a query using the Class Editor, position the cursor on a blank line in the Class Editor and enter a query declaration:

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;

/// This query provides a list of persons ordered by Name.
Query ByName(ByVal name As %String) As %SQLQuery(CONTAINID = 1)
{
    SELECT ID,Name FROM Person
    WHERE (Name %STARTSWITH :name)
    ORDER BY Name
}

}
```

Alternatively, you can copy and paste an existing query declaration and then edit it.

For a general introduction to queries, see the chapter “Queries” of *Using Caché Objects*. For details on query definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*.

9.1 New Query Wizard

You can use the New Query wizard to add a new query to a class definition. You can open the New Query wizard using **Class > Add > Query**. Alternatively, right-click the Class Inspector and select **Add Query** or select the **New Query** icon, , in the toolbar.

Select **Finish** at any time; default values are provided for any information you have not specified.

9.1.1 Name, Implementation, and Description Page

The New Query wizard prompts you for the following information (you can later modify any of these values):

Query Name

(required) Name of the new query. This name must be a valid query name and must not conflict with the name of a previously defined query.

For a discussion of names, see the chapter “Caché Classes ” in *Using Caché Objects*.

Implementation

(required) You must specify if this query is based on an SQL statement (which the wizard generates for you) or on user-written code (in which case you have to provide the code for the query implementation).

Description

(optional) Description of the new query. This description is used when the class documentation is displayed in the online class library documentation.

A description may include HTML formatting tags. For more details, see “Using HTML Markup in Class Documentation” in the chapter “Defining and Compiling Classes” in *Using Caché Objects*.

9.1.2 Input Parameters Page

A query may take zero or more input parameters (arguments).

You can specify the names, types, default values, and how data is passed by value for these parameters. The arguments are displayed in order in a table. You can add a new item to the argument list using the **Add** icon located on the side of the table. This displays a popup dialog allowing you to specify the name of the argument, its type, its optional default value. Using up, ⬆, and down arrows, ⬇, you can rearrange the order of items in the list.

9.1.3 Columns Page

For an SQL-based query, you must specify the object properties (columns) that you want included in the result set (this is the SELECT clause of the generated SQL query).

To add a column to the query, select an item from the left-hand list of available properties and move it to the right-hand list using the **>** (Move To) button.

9.1.4 Conditions Page

For an SQL-based query, you can specify conditions to restrict the result set (the SQL WHERE clause of the generated SQL query).

You can build a set of conditions by selecting values from the set of combo boxes. The expression box can contain an expression (such as a literal value) or a query argument (as an SQL host variable with a prepended : colon character).

9.1.5 Order By Page

For an SQL-based query, you can specify any columns you want to use to sort the result set (the SQL ORDER BY clause of the generated SQL query).

9.1.6 Row Specification Page

For a user-written query, you must specify the names and types of the columns that are to be returned by the query.

The wizard does not prompt for this information for SQL-based queries as the class compiler can determine it by examining the SQL query.

9.1.7 Results of Running the New Query Wizard

After you run the New Query wizard, the Class Editor window is updated to include the new query definition. For example:

```
/// This is a Person class
class MyApp.Person extends %Persistent
{

Query ByName(ByVal name As %String) As %SQLQuery(CONTAINID = 1)
{
    SELECT ID,Name FROM Person
    WHERE (Name %STARTSWITH :name)
    ORDER BY Name
}

}
```

If you want to make further modifications to this query you can do this using either the Class Editor or the Class Inspector.

If you specified a user-written query, the Class Editor contains both the new query definition as well as skeletons of the query methods you are expected to implement:

```
Class MyApp.Person Extends %Persistent
{
// ...

ClassMethod MyQueryClose(
        ByRef qHandle As %Binary
        ) As %Status [ PlaceAfter = MyQueryExecute ]
{
    Quit $$$OK
}

ClassMethod MyQueryExecute(
        ByRef qHandle As %Binary,
        ByVal aaa As %Library.String
        ) As %Status
{
     Quit $$$OK
}

ClassMethod MyQueryFetch(
        ByRef qHandle As %Binary,
        ByRef Row As %List,
        ByRef AtEnd As %Integer = 0
        ) As %Status [ PlaceAfter = MyQueryExecute ]
{
    Quit $$$OK
}

Query MyQuery(
        ByVal aaa As %Library.String
        ) As %Query(ROWSPEC = "C1,C2")
{
}
}
```

10

Adding Indices to a Class

This chapter discusses how to add and edit index definitions to a persistent class definition.

An index definition instructs the Caché class compiler to create an index for one or more properties. Indices are typically used to make SQL queries more efficient.

You can add an index to a class definition in two ways:

- Editing the class definition using the Class Editor.
- Using the New Index wizard

To add an index using the Class Editor, position the cursor on a blank line in the Class Editor and enter an index declaration:

```
Index NameIndex On Name;
```

Alternatively, copy and paste an existing index declaration and then edit it.

For details on index definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*.

10.1 New Index Wizard

You can invoke the New Index wizard using the **Class > Add > Index**. Alternatively, right-click the Class Inspector and select **Add Index** or select the **New Index** icon, , from the toolbar.

The New Index wizard prompts you for information. To end, select **Finish** (default values are provided for any information you have not specified).

10.1.1 Name and Description Page

The New Index wizard prompts you for the following information (you can later modify any of these values):

Index Name

(required) Name of the new index. This name must be a valid index name and must not conflict with the name of a previously defined index.

For a discussion of valid names, see the chapter “Caché Classes” in *Using Caché Objects*.

Description

(optional) Description of the new index. This description is used when the class' documentation is displayed in the online class library documentation.

A description can include HTML formatting tags. For details, see "Using HTML Markup in Class Documentation" in the chapter "Defining and Compiling Classes" in *Using Caché Objects*.

10.1.2 Index Type Page

Caché supports the following types of indices.

Normal Index

A normal index is used for indexing on property values. You can further qualify a normal index by selecting one of the following:

Unique Index	The set of properties associated with this index must have a combined value that is unique in the extent of objects of this class.
IDKEY	The set of properties associated with this index are used to create the Object ID value used to store instances of this class in the database. You cannot modify the values of properties that are part of an IDKEY definition once an object has been saved. IDKEY implies that the property or properties are unique (as with a Unique Index).
SQL Primary Key	The set of properties associated is reported as the SQL Primary Key for the SQL table projected for this class. This implies that the property or properties are unique (as with a Unique Index).

Extent Index

An extent index is used to keep track of which objects belong to a specific class in a multiclass extent of objects. It differs from a so-called normal index in that you cannot specify additional characteristics for it.

You can also select how the index is physically implemented in the database:

Standard Index

This index is a traditional cross-index on the specified property or properties.

Bitmap Index

A bitmap index uses a compressed representation of a set of object ID values that correspond to a given indexed value. See Bitmap Indices in *Caché SQL Optimization Guide* for more information.

Bitslice Index

A bitslice index is a specialized form of index that enables very fast evaluation of certain expressions, such as sums and range conditions. Bitslice indices are currently automatically used in certain Caché SQL queries. Future versions of Caché SQL will make further use of bitslice indices to optimize additional queries.

10.1.3 Index Properties Page

On the Index Properties page, you can enter a list of one or more properties on which the index is based. For each property you can override the default collation function used to transform values stored in the index as well as any parameters for the collation function.

Note: Important — There must not be a sequential pair of vertical bars (||) within the values of any property used by an IDKEY index, unless that property is a valid reference to an instance of a persistent class. This restriction is imposed by the way in which the Caché SQL mechanism works. The use of || in IDKey properties can result in unpredictable behavior.

10.1.4 Index Data Page

On the Index Data page, elect to store a copy of the data for any properties in the index.

You cannot store copies of data values with a bitmap index.

10.1.5 Results of Running the New Index Wizard

After running the New Index wizard, the Class Editor window is updated to include the new index definition. For example:

```
/// This is a Person class
class MyApp.Person extends %Persistent
{

Property Name As %String;

Index NameIndex On Name;
}
```

You can edit the index definition with either the Class Editor or the Class Inspector.

10.2 Populating an Index

After you add an index definition to a class and compile, you can populate the index (place data into it) by using **Rebuild Indices**, found in the Management Portal (Building Indices).

The Studio does not automatically place data into indices.

11

Adding Projections to a Class

This chapter discusses how to add projection definitions in a class definition.

A projection definition instructs the Caché class compiler to perform specified operations when a class definition is compiled or removed. A projection defines the name of a projection class (derived from the %Projection.AbstractProjection class) that implements methods that are called when a) the compilation of a class is complete and b) when a class definition is removed (either because it is being deleted or because the class is about to be recompiled).

A class can contain any number of projection definitions. The actions for all of them are invoked when the class is compiled (the order in which they are invoked is not defined).

Caché includes predefined projection classes that generate client code that allows access to a class from Java, MV, and so on. See the class in the class reference for definitions of parameters for each class.

To generate C++-related classes, use Tools > Generate C++ Projection or the **cpp_generator** command line interface. For more information, see the chapter "Using the C++ Generator Program" in *Using C++ with Caché*.

Table 11–1: Projection Classes

Class	Description
%Projection.Java	Generates a Java client class to enable access to the class from Java.
%Projection.Monitor	Registers this class as a routine that works with Caché Monitor. Metadata is written to Monitor.Application, Monitor.Alert, Monitor.Item and Monitor.ItemGroup. A new persistent class is created called Monitor.Sample.
%Projection.MV	Generates an MV class that enables access to the class from MV.
%Projection.StudioDocument	Registers this class as a routine that works with Studio.
%Studio.Extension.Projection	Projects the XData 'menu' block to the menu table.
%ZEN.Object.Projection	Projection class used by %ZEN.Component.object classes. This is used to manage post-compilation actions for Zen components.
%ZEN.PageProjection	Projection class used by %ZEN.Component.page. Currently this does nothing.
%ZEN.Template.TemplateProjection	Projection class used by %ZEN.Templage.studioTemplate class.

You can also create your own projection classes and use them from Studio as you would any built-in projection class.

You can add a new projection to a class definition in two ways:

- Editing the class definition using the Class Editor.
- Using the New Projection wizard

To add a projection using the Class Editor, position the cursor at a blank line and enter a projection declaration.

Alternatively, you can copy and paste an existing projection declaration and then edit it.

For details on projection definitions, see “Class Definitions” in the reference “Class Definitions” in *Caché Class Definition Reference*.

11.1 New Projection Wizard

You can invoke the New Projection wizard using the **Class > Add > Projection** and asking for a new Projection. Alternatively right-click in the Class Inspector and select **New Projection**.

The New Projection wizard displays pages prompting you for information about the new projection. To end, select **Finish** (in this case, default values are provided for any information you have not specified).

11.1.1 Name and Description Page

The New Projection wizard prompts you for the following information (you can later modify any of these values):

Projection Name

> (required) Name of the new projection. This name must be a valid projection name and must not conflict with the name of a previously defined projection.

Description

> (optional) Description of the new projection.

11.1.2 Projection Type Page

The projection type determines what actions happen when your class definition is compiled or removed. You can select what kind of projection you would like to define:

Projection Type

> Name of a projection class whose methods are executed when a class definition is compiled or removed.

Projection Parameters

> A set of name-value pairs that control the behavior of the projection class. The list of available parameter names is determined by the selected projection class.

11.1.3 Results of Running the New Projection Wizard

When you finish running the New Projection wizard, the Class Editor window is updated to include the new projection definition. For example:

```
/// This is a Person class
class MyApp.Person extends %Persistent
{

Property Name As %String;

Projection JavaClient As %Projection.Java;
}
```

To edit this projection definition, use either the Class Editor or the Class Inspector.

12

Adding XData Blocks to a Class

An XData block is a block of XML code that you can add to your class definition.

You can add an XData block to a class definition in two ways:

- Editing the class definition using the Class Editor.
- Using the XData wizard

To add an XData block using the Class Editor, position the cursor at a blank line and enter an XData declaration:

```
XData ProductionDefinition
    {
    <Production>
    <ActorPoolSize2/ActorPoolSize>
    </Production>
    }
```

Alternatively, you can copy and paste an existing XData block and then edit it.

12.1 New XData Wizard

You can invoke the New Projection wizard using the **Class > Add > XData**. Alternatively, right-click in the Class Inspector and select **Add Projection**.

The New XData wizard displays a single page prompting you for a name for the XData block and a description. To end, select **Finish**. Add XML code into the Class Editor window to complete the XData block.

The New XData wizard prompts you for the following information (you can later modify any of these values):

XData Name

(required) Name of the new XData. This name must be a valid name and must not conflict with the name of a previously defined XData.

Description

(optional) Description of the new XData.

13

Adding SQL Triggers and Foreign Keys to a Class

Every persistent Caché class is automatically projected as an SQL table. This chapter discusses how you can use Studio with those parts of a class definition that control its SQL behavior.

13.1 SQL Aliases

You can give classes as well as most class members an alternate name for use by SQL. This is useful because:

- There is a long list of SQL reserved words that cannot be used as identifiers.
- Caché does not support the underscore character in class or class member names.

To specify an SQL table name for a class, view the Class information in the Class Inspector and edit the value for the `SqlTableName` keyword.

To specify an SQL name for a class member, select the desired property in the Class Inspector and edit the value for its appropriate SQL name keyword (such as `SqlFieldName` for properties and `SqlName` for indices).

13.2 SQL Stored Procedures

An SQL Stored procedure is a Caché method or class query than can be invoked from an ODBC or JDBC client as a stored procedure.

Caché supports two styles of SQL stored procedure: those based on class queries and that return a result set; and those based on class methods and that do not return a result set.

13.2.1 Query-Based Stored Procedure

To create an SQL stored procedure that returns a result set, add a query definition to a class definition and then set the query's SqlProc keyword to true. Do this as follows:

1. Create a query in a class definition using the New Query wizard with **Class > Add > Query**.
2. Using the Class Inspector, set the value of the query definition keyword SqlProc to `True`.

You should end up with something similar to:

```
Class Employee Extends %Persistent
{

/// A class query listing employees by name.
Query ListEmployees() As %SQLQuery(CONTAINID = "1") [SqlProc]
{
    SELECT ID,Name
    FROM Employee
    ORDER BY Name
}
}
```

You can invoke this stored procedure from an ODBC or JDBC client using a **CALL** statement:

```
CALL Employee_ListEmployees()
```

Following this call, the ODBC or JDBC application can fetch the contents of the result set returned by the class query.

Note that you can use this same technique with query definitions that are based on custom-written code; you are not limited to defining stored procedures solely based on SQL statements.

13.2.2 Creating Method-Based Stored Procedure

To create an SQL stored procedure that does not return a result set, add a class method to a class definition and then set the method's keyword SqlProc to `True`. Do this as follows:

1. Create a class method in a class definition using the New Method wizard.
2. Using the Class Inspector, set the value of method's keyword SqlProc to `True`.

You should end up with something similar to:

```
Class Employee Extends %Persistent
{

  ClassMethod Authenticate(
        ctx As %SQLProcContext,
        name As %String,
        ByRef approval As %Integer
        ) [SqlProc]
  {
    // ...
    Quit
  }

}
```

Note that the first argument of a method used as an SQL stored procedure is an instance of a %SQLProcContext object. For more information, see the chapter “Defining and Using Stored Procedures” in *Using Caché SQL*.

You can invoke this stored procedure from an ODBC client using a **CALL** statement:

```
CALL Employee_Authenticate('Elvis')
```

To invoke this stored procedure from a JDBC client, you can use the following code:

```
prepareCall("{? = call Employee_Authenticate(?)}")
```

13.3 Adding SQL Triggers to a Class

An SQL trigger is code that is fired by the SQL Engine in response to certain events.

Note that SQL triggers are not fired during object persistence (unless you are using %CacheSQLStorage storage class).

You can add an SQL trigger to a class definition in two ways:

- Editing the class definition using the Class Editor.
- Using the New SQL Trigger wizard

To add an SQL trigger using the Class Editor, position the cursor on a blank line in the Class Editor and enter a trigger declaration:

```
Class MyApp.Company Extends %Persistent
{

/// This trigger updates the Log table for every insert
Trigger LogEvent [ Event = INSERT ]
{
    // ...
}

}
```

13.3.1 New SQL Trigger Wizard

You can use the New Trigger wizard to create a new SQL trigger. You can open the New SQL Trigger wizard using **Class > Add > SQL Trigger**. Alternatively, right-click in the Class Inspector and select **Add SQL Trigger** .

The New SQL Trigger wizard prompts you for information. Select **Finish** at any time (default values are provided for any information you have not specified).

13.3.1.1 Name and Description Page

The New SQL Trigger wizard prompts you for the following information (you can later modify any of these values):

Trigger Name

(required) Name of the trigger. This name must be a valid trigger name and must not conflict with the name of a previously defined trigger.

Description

(optional) Description of the new trigger. This description is used when the class' documentation is displayed in the online class library documentation.

13.3.1.2 Trigger Event Page

The New SQL Trigger wizard asks you to indicate when you want the new trigger to be fired by specifying the event and time for the trigger.

Event Type

This specifies which SQL event fires the trigger. The choices are `Insert` (when a new row is inserted), `Update` (when a row is updated), or `Delete` (when a row is deleted).

Event Time

This specifies when the trigger is fired. The choices are `Before` or `After` the event occurs.

13.3.1.3 Trigger Code

The New SQL Trigger wizard lets you enter the source code for the trigger if you want.

13.3.1.4 Results of Running the New SQL Trigger Wizard

After you finish using the New SQL Trigger wizard, the Class Editor window is updated to include text for the new trigger definition.

If you want to edit this trigger you can do this using either the Class Editor or the Class Inspector.

13.4 Adding New SQL Foreign Keys to a Class

An SQL foreign key defines an integrity constraint between one or more fields in a table and a key (unique index) in another table.

Object applications typically do not use foreign keys; they instead use relationships which offer better object-based navigation. Relationships automatically impose integrity constraints (for both SQL and object access) that are equivalent to manually defining foreign key definitions.

Typically you use foreign key definitions in applications that are originally purely relational in nature.

You can add an SQL foreign key to a class definition in two ways:

- Editing the class definition using the Class Editor.
- Using the New SQL Foreign Key wizard

To add an SQL foreign key using the Class Editor, position the cursor on a blank line in the Class Editor and enter a foreign key declaration:

```
Class MyApp.Company Extends %Persistent
{

Property State As %String;

ForeignKey StateFKey(State) References StateTable(StateKey);

}
```

13.4.1 New SQL Foreign Key Wizard

Open the New SQL Foreign Key wizard using the **Class > Add > Foreign Key**. Alternatively you can right-click the Class Inspector and selecting **Add Foreign Key** or select the **New Foreign Key** icon, , from the toolbar.

The New SQL Foreign Key wizard prompts you for information. When you have filled in the required information, select **Finish** (default values are provided for any information you have not specified).

13.4.1.1 Name and Description Page

The New SQL Foreign Key wizard prompts you for the following information (you can later modify any of these values):

Foreign Key Name

(required) Name of the foreign key. This name must be a valid foreign key name and must not conflict with the name of previously defined foreign key.

For a general discussion on names see the chapter “Caché Classes ” in *Using Caché Objects*.

Description

(optional) Description of the new foreign key. This description is used when the class' documentation is displayed in the online class library documentation.

A description can include HTML formatting tags. For more details, see "Using HTML Markup in Class Documentation" in the chapter "Defining and Compiling Classes" in *Using Caché Objects*.

13.4.1.2 Attributes Page

The second page asks you to select one or more properties of the class that you want constrained by the foreign key.

13.4.1.3 Key Construction Page

The third page asks you to select both the class and a key (unique index) in that class that specify the values used to constrain the foreign key properties.

13.4.1.4 Results of Running the New SQL Foreign Key Wizard

After running the New SQL Foreign Key wizard, the Class Editor window is updated to include the new foreign key definition.

If you want to make further modifications to this foreign key you can do this using either the Class Editor or the Class Inspector.

14

Adding Storage Definitions to a Class

The physical storage used by a persistent or serial class is specified by means of a *storage definition*. You can use Studio to view and edit such storage definitions.

Note: Storage definitions are a fairly advanced feature of Caché objects. In most cases, you do not need to work with storage definitions; the Caché class compiler automatically creates and manages storage definitions for persistent objects.

If you use storage definitions, you typically work with them in the following cases:

- You need detailed control over the storage used by a persistent object, perhaps for performance tuning.
- You are mapping an object definition on top of a preexisting data structure.

A class can have any number of storage definitions, though only one can be used at one time. A new class does not have a storage definition until either you first a) save and compile the class, or, b) you explicitly add one. You can add a new storage definition to a class using **Class > Add > Storage**.

Note: Compiling a class automatically generates its storage definition. Only persistent and serial classes have storage definitions.

Within Studio, you can view and edit the storage definition(s) for a class in two different ways:

- *Visually* using the Storage Inspector in the Class Inspector window: select `Storage` in the Class Inspector and select the desired storage definition.
- *Textually* using the Class Editor window; the storage definition is in the body of the class definition.

These techniques are described in the following sections.

14.1 Adding Storage Definitions to a Class

You can add a new storage definition to a class definition in two ways:

- Adding a storage definition to the class definition using the Class Editor and **Class > Add > Storage**.
- Using the New Storage wizard.

14.1.1 Using the New Storage Wizard

You can use the New Storage wizard to add a new storage definition to a class definition. You can start the New Storage wizard using **Class > Add > Storage**. Alternatively, right-click in the Class Inspector and selecting **Add > Storage** or select the **New Storage** icon, , from the toolbar.

The New Storage wizard prompts you for information. Select **Finish** at any time (in this case, default values are provided for any information you have not specified).

14.1.1.1 Name, Type, Description Page

The New Storage wizard prompts you for the following information (you can later modify any of these values):

Storage Name

(required) Name of the new storage definition. This name must be a valid class member name and must not conflict with the name of a previously defined storage definition.

Storage Type

(required) Type of storage used by this storage definition. The type specifies which storage class is responsible for implementing the storage interface for this class. The choices are:

- Caché Storage—this storage definition is based on the %CacheStorage class. This is the default storage type used for new persistent classes.
- Caché SQL Storage—this storage definition is based on the %CacheSQLStorage class. This storage type uses SQL statements to perform storage operations. This storage type is used for mapping objects to existing data structures or to talk to remote RDBMS via the Caché SQL Gateway.
- Custom Storage—this storage definition is based on a user-defined storage class.

Description

(optional) Description of the new storage definition.

A description can include HTML formatting tags. For more details, see "Using HTML Markup in Class Documentation" in the chapter "Defining and Compiling Classes" in *Using Caché Objects*.

14.1.1.2 Global Characteristics of a %CacheStorage Definition Page

For a %CacheStorage storage definition, the New Storage wizard lets you specify some characteristics of the globals (persistent multidimensional arrays) used to store the data and indices for the persistent class. These characteristics include:

DataLocation

Name of the global as well as any leading subscripts used to store instances of the persistent class. For example, to specify that data should be stored in the global *^data*, enter `^data` in this field. To specify that data should be stored in the global subnode *^data("main")*, enter `^data("main")`.

IndexLocation

Name of the global as well as any leading subscripts used to store index entries for the persistent class. By default, indices are stored in the Index Reference global with an additional subscript based on the Index name. You can override this on an index-by-index basis.

IdLocation

Name of the global as well as any leading subscripts used to contain the default object ID counter. The object ID counter is used to maintain the ID number of the next object instance of this type.

14.2 Using the Class Inspector with Storage Definitions

You can use the Class Inspector to visually view and edit a class' storage definition. The Class Inspector displays a list of storage definitions in the same way that it displays methods or properties.

To view an existing storage definition in the Class Inspector:

1. Select the Class Inspector
2. Select `Storage` in the Inspector's Member list pull-down.
3. Double-click a storage definition.

At this point, the Class Inspector displays storage keywords along with their values.

Several of the storage keywords warrant special attention:

Data Nodes

Represents the set of data mappings used by the %CacheStorage storage class. The Data Nodes editor, which you can invoke by double-clicking on the `Data Nodes` keyword, allows you to view and edit the set of data node specifications for the storage definition: that is, you can directly specify how your class' properties are stored in global nodes.

SQL Storage Map

Represents the set of data mappings used by the %CacheSQLStorage storage class. The SQL Storage Map editor, which you can invoke by double-clicking the `SQL Storage Map` keyword, allows you to view and edit the set of mappings used to map object properties to existing data structures.

14.3 Using the Class Editor with Storage Definitions

You can use the Class Editor to view and edit a class' storage definition. You can toggle the display of a class storage definition(s) using **View > Expand Code**.

A storage definition is displayed as an in-line XML island in the body of the class definition using the same XML elements that are used in the external, XML-representation of a class definition.

For example, suppose you have a simple persistent MyApp.Person class:

```
/// A simple persistent class
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property City As %String;
}
```

After compiling this class (to ensure that the class compiler has created a storage definition for it), display its storage definition using the **View > Expand Code** (or select plus icon next to the top line of the block). This results in following display in the Class Editor window:

```
/// A simple persistent class
Class MyApp.Person Extends %Persistent
{

Property Name As %String;
Property City As %String;

<Storage name="Default">
    <Data name="PersonDefaultData">
        <Value name="1">
            <Value>City</Value>
        </Value>
        <Value name="2">
            <Value>Name</Value>
        </Value>
    </Data>
    <DataLocation>^MyApp.PersonD</DataLocation>
    <DefaultData>PersonDefaultData</DefaultData>
    <IdLocation>^MyApp.PersonD</IdLocation>
    <IndexLocation>^MyApp.PersonI</IndexLocation>
    <Type>%Library.CacheStorage</Type>
</Storage>
}
```

The XML storage definition includes all the defined storage keywords and their corresponding values represented as XML elements.

You can directly edit this definition in the Class Editor as you would any other part of the class definition. If you enter invalid XML syntax the editor colors it as an error.

The storage definition can be useful in cases where you need to do simple or repetitive modifications.

For example, suppose you want to change the name of a property City to HomeCity, while preserving the physical storage layout (that is, you want the new property name to access the values stored with the old name). You can do this using the Class Editor as follows:

1. Load the class definition into a Studio Class Editor window and display its storage.
2. Use the Editor's **Replace** command to replace all occurrences of the property City with HomeCity. You must be careful to only change those occurrences of City that represent the property name (such as in the property definition, method code, descriptions, and in the body of the storage definition); do not replace the values of any class definition keywords.
3. Save and recompile the class definition.

15

Working with CSP Files

A CSP (Caché Server Page) file is a text file containing HTML, XML, or CSP markup commands. This file is stored on a Caché server machine and is compiled, by the CSP Engine, into an executable class that can process HTTP events sent from a browser.

You can use Studio to create and edit CSP files in the same way you would work with class definitions or routines. CSP files are displayed in the Studio syntax-coloring editor allowing you to quickly spot errors in HTML as well as in any embedded server-side scripts.

15.1 Sample CSP Page

To create a simple CSP page with Studio, perform the following steps:

1. Start Studio and create a new Project in the SAMPLES namespace.
2. Create a new CSP file (with **File > New > CSP File tab > Caché Server Page**) .
3. Studio creates a new CSP Editor window containing a new CSP file named Untitled.csp.

   Replace the contents of the editor window with:

   ```
   <HTML>
   <HEAD>
   <TITLE>Sample Page</TITLE>
   </HEAD>
   <BODY>
   My Sample CSP Page.
   </BODY>
   </HTML>
   ```

4. Save the page with **File > Save**. The **Save As** dialog appears. Within the dialog, double-click the CSP application /csp/samples (this is the directory in which we are going to save this CSP page) and then enter Sample.csp in the **Filename** box. Select **Save As**.
5. Compile the page with **Build > Compile** .
6. View the resulting Web page from a browser with **View > Web Page**.

At this point, you should see a very simple Web page containing the words `My Sample CSP Page` in your browser.

To make this example more interesting, we can add an SQL query to the page that executes when the page is requested:

1. Position the cursor in the CSP Editor window at the start of the blank line after `My Sample CSP Page`.
2. Select **Insert > SQL Query**. In the dialog that appears enter the following SQL query:

```
SELECT Name,SSN FROM Sample.Person ORDER BY Name
```

3. Check **Create HTML Table** and select **OK**.
4. Save and recompile the page with **Build > Compile**.
5. View the resulting Web page from a browser with **View > Web Page** .

Now your CSP page displays a list of names and social security numbers in an HTML table.

15.2 Creating a New CSP File

To create a new CSP file, select **File > New > CSP File tab > Caché Server Page**. This creates a new CSP file named Untitled.csp.

When you save this file for the first time, you are asked for a file name. If this file is part of a CSP application, create a folder with an application name, in which to put your new file.

The file name, which must have a .csp extension, is used for both saving a physical source file on the Caché server as well as in a URL requesting this page. The application name also determines the URL used to request the CSP page as well as other characteristics.

For more information on CSP files and applications, see the Caché Server Pages documentation.

15.2.1 Default.csp Template File

When you create a new CSP file in Studio, it opens a new CSP Editor window and copies into it the contents of a CSP template file. You can edit or replace this template file in order to customize how Studio creates new CSP files. This file is a text file called Default.csp and is located in the same directory as the Studio executable file. For a default installation, this is the /cachesys/bin directory.

15.3 Editing a CSP File

You can edit a CSP file in the same way you would edit any other document in Studio.

15.3.1 Insert Options

The Studio includes dialogs, a wizard, and templates to assist with editing CSP files. These dialogs are available under the **Insert** menu and are described in the table below.

The templates are in theCSP/samples directory in the installation directory . To view a sample, open a file in Studio, such as zipcode.csp and then select **View > Web Page**. The Web Form Wizard is available in the **Tools > Templates > Templates** menu.

Insert Menu Option	Action
Class	Inserts a <csp:CLASS> tag at the current cursor location.
Loop	Inserts a <csp:LOOP> tag at the current cursor location.
While	Inserts a <csp:WHILE> tag at the current cursor location.
Method	Inserts a Caché objects method (in a <SCRIPT> tag) at the current cursor location
Object	Inserts a <csp:OBJECT> tag at the current cursor location.
Query	Inserts a <csp:QUERY> tag at the current cursor location.
SQL Query	Inserts an SQL query (in a <SCRIPT> tag) at the current cursor location.

15.4 Saving a CSP File

Save a CSP file using the **File > Save**. This sends the source of the CSP file back to the Caché server (which could be on a remote machine) and save it on the server's local file system in the appropriate directory (specified by the Caché server's CSP application settings). Studio automatically saves backup files for the five previous saves of a CSP file. For more information, see Save Automatically Backs Up Routines, Include, and CSP Files

15.5 Compiling a CSP File

Compile a CSP file using **Build > Compile**. Compiling a CSP File is a multi-step process: first the CSP file is fed through the CSP engine and converted into a Caché class (derived from the %CSP.Page class). Then this generated class is compiled into one or more routines that contain executable code.

Sometimes it is easier to debug or understand a CSP file by looking at the code generated for it. You can use Studio to view the class generated for a CSP file, as well as the routine(s) generated from this class by opening them with **File > Open** or **View > Other**.

15.6 Viewing the Results of a CSP File

You can view the results of a CSP file in a browser by using **View > Web Page**. This launches your default browser with the URL for the current CSP page. You can also use this command when editing a %CSP.Page class.

You can modify the server address portion of the URL used to display a CSP page in a specific project. To do this, select **Project > Settings** and edit the value of the `WEB Server` field.

15.7 Viewing Syntax-Colored Source for Any URL

As an aid to debugging Web applications, Studio lets you request a Web page from a URL and display its HTML source in a syntax coloring window. This can help you spot errors in Web pages more easily than viewing the rendered HTML in a browser.

You can open a URL Viewer window using the **File > Open URL** and entering a URL in the resulting dialog.

To try this with a CSP sample page, do the following:

1. Select **File > Open URL**.
2. If you have a Web server on your local system, enter http://localhost/csp/samples/custom.csp

   If you do not have a Web Server on your local system, use the test HTTP server on the 8972 port (or the port number your system is configured for): http://localhost:8972/csp/samples/custom.csp

At this point, you sees the HTML returned by the custom.csp page displayed in a syntax-coloring window.

Note: You can use the URL viewer to view syntax-colored source for any Web page on the Internet.

16

Working with Routines and Include Files

A routine is the unit of execution in a Caché server; all application logic running on a Caché server is executed by invoking routines and entry points in routines. Routines are executed in a virtual machine that is built into the Caché server environment. Routines are portable to all platforms supported by Caché and automatically shareable across a Caché environment.

Include files (.inc files) contain macro definitions (or other include files) and can be included in .mac routines or class definitions. For more information on macros see the section "Using Macros" in *Using Caché ObjectScript*.

16.1 Routine Editor

Using the Routine Editor, you can directly create and edit the source for routines or include files. When class definitions are compiled, the class compiler generates a set of routines containing the implementation for the class. If you check to **Keep Generated Source Code** (on the **Tools > Options** dialog, **Compiler, General Flags** tab), you can view and edit this generated source code as well.

The Routine Editor uses syntax coloring and indicates syntax errors with a wavy red line.

16.2 Routine Source Formats

There are several kinds of routine source formats (files) in Caché. The Routine Editor provides syntax coloring and checking for each of these formats. The formats include:

- BAS - Basic source files with a .bas extension
- MAC - Macro source files with a .mac extension, processed by the Caché macro preprocessor to resolve macros, embedded SQL statements, and embedded HTML, which results in an .int file.
- INT - Intermediate source files, which are compiled directly into executable Caché object (OBJ) code.
- INC - Include files. Not routines per se, .inc files contain macro definitions that can be included by .mac routines.

By default, when you create a new ObjectScript routine, it is saved as a .mac routine. Select **File > Save As** to save this as a different type of routine (changing the extension from .mac to .inc for example).

Select **View > View Other** to display .int code corresponding to a given .mac file and vice versa.

16.3 Creating a New Routine or Include File

To create a new routine or include file, select **File > New**. A dialog displays the templates you can choose from. For a routine, you can choose either an ObjectScript routine or a Caché Basic routine. For an include file, select ObjectScript. This opens a new Routine Editor window with a default name, such as Untitled. You can save this with a different name with **File > Save As**.

16.4 Opening an Existing Routine or Include File

Open an existing routine with **File > Open** . In the drop-down list of **Files of Type** , select the file extension of interest (such as .mac, .int, or **All Files**) and select a routine.

When you attempt to open a previously saved routine or include file, the **Open** dialog uses wildcard matching (using the * (asterisk to match any number of any character) and ? (question mark to match a single character) to display a list of available routines or include files. The routine type - BAS, MAC, INT, or INC - is used as a file extension for purposes of wildcard matching.

16.5 Routine Template File

When you create a new routine in Studio, it opens a new Routine Editor window. If a Routine template file exists, it is copied into the new file. To create a Routine template file, create a file with the contents that you want in your template. Save the file asDefault.mac in the same directory as the Studio executable file (CStudio.exe).

16.6 Saving, Compiling, and Deleting Routines

You can save routines to the database using the **File > Save** or **File > Save As**. By default, saving a routine does not cause it to be compiled. You can change this behavior by checking **Compile Routine on Save** available from **Tools > Options** dialog, **Compiler, Behavior**.

To compile a routine directly, use **Build > Compile** (which also causes it to be saved).

To delete a routine, in a Workspace window, highlight the routine and select **Edit > Delete**. The routine and any generated files are deleted.

16.7 Save Automatically Backs Up Routines, Include, and CSP Files

When you save an existing routine (or include or CSP file), Studio automatically creates a backup file. It automatically saves up to five backup files, naming them with a ;# (semicolon number) suffix. For example, a file named setup.MAC which has been saved six times has five backup files named:

```
setup.MAC;1
setup.MAC;2
setup.MAC;3
setup.MAC;4
setup.MAC;5
```

Specifically, files with the following extensions are automatically backed up: .BAS, .INC, .INT, .MAC, .OBJ, .MVB, .MVI, .CSP

To see what backup files exist, use a semi-colon in the search field of the **File > Open** option. You can use the following syntax examples:

.;* displays all backup files in this folder

.mac; displays all backup files with a .MAC extension

setup.*;* lists all backup files named setup

These backup files can also be found using this syntax through the Management Portal on the **System Explorer** > **Routines** page using the **Routines and include files** box.

17

Using the Studio Debugger

The Studio debugger lets you step through the execution of programs running on a Caché server. Programs that can be debugged include INT files, BAS files, MAC files, methods within CLS files, CSP classes responding to HTTP requests, server-side methods invoked from Java or ActiveX clients, or server-hosted applications. To step through, or set breakpoints within classes or CSP pages, open the corresponding INT or BAS file and use the debugging commands in it. To view INT source code files, go to the **Tools > Options** dialog, **Compiler, General Flags** tab and enable the `Keep Generated Source Code` option.

You can connect the debugger to a target process in one of the following ways:

- Define a debug target (name of program, routine, or Zen or CSP page to debug) for the current project using **Project > Settings > Debugging > Debug Target** (or **Debug > Debug Target**). Then select **Debug > Go** to start the target program and connect to its server process.
- Select **Debug > Attach** and choose a running process on a Caché server.

Sometimes using command-line debugging with the **zbreak** command can give you better control. For more information on **zbreak**, see the chapter "Command-Line Routine Debugging" in *Using Caché ObjectScript*.

17.1 Sample Debugging Session: Debugging a Routine

The following example demonstrates how to debug a Caché routine.

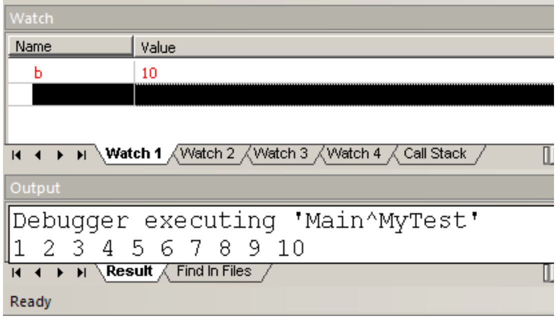
1. Start Studio and select **File > New Project** to create a new project called **Project1**.
2. Create a new routine by selecting **File > New > General tab > ObjectScript Routine**.
3. Enter code for this routine:

   ```
   MyTest ; MyTest.MAC

   Main() PUBLIC {
       Set a = 10
       For i = 1:1:10 {
           Set b = i
                Write b," "
       }
   }
   ```

4. Save and compile the new routine as MyTest.MAC using **File > Save As**.
5. Define a debug target for the project by selecting the **Debug > Debug Target** tab, selecting **Class Method or Caché Routine**, and entering the name of the entry point in your new routine, `Main^MyTest`.

6. Set a breakpoint in the routine: Position the cursor anywhere on the line `Set a = 10` and press **F9**, the Toggle Breakpoint key. A breakpoint indicator appears in the left margin, ●.
7. Select **Debug > Go** to begin debugging. When the debugger stops at your breakpoint, the next command to be executed is outlined with a yellow box. The INT file opens in a new window (if you enabled the `Keep Generated Source Code` option, on the **Tools > Options** dialog, **Compiler, General Flags** tab).
8. Enter `b` and `a` (as *Watchpoints*) in the Watch window (**View > Watch**) so that you can watch the values.
9. Step through execution of the program by repeatedly selecting **Debug > Step Into** (F11) and notice the `b` value change.

You can stop debugging by stepping to the end of the program or by selecting **Debug > Stop**.



Name	Value
b	10

```
Debugger executing 'Main^MyTest'
1 2 3 4 5 6 7 8 9 10
```

17.2 Debugger Settings for the Current Project

Some debug settings are defined and stored in the current project. These include:

- Debug Target
- Breakpoints

17.2.1 Debug Target

A debug target tells Studio what process you want to debug.

To specify a debug target for a project, select **Project > Settings > Debugging > Debug Target** or select **Debug > Debug Target**. Choose one of the following, which is started when you select **Debug > Go**. You can also set a debug target by placing the cursor next to an item in a editor window, right-clicking, and selecting **Set xxxx as debug target**.

Class Method or Caché Routine

The routine (and tag), class, or method that you want to debug when **Debug > Go** is executed. For example, enter `Test^MyRoutine()` to begin execution at the tag `Test` in the routine `MyRoutine`. Or enter the name of a class method to execute, such as `##class(MyApp.Person).Test(1)`.

ZEN and CSP Page (URL, CSP or class)

The Zen or CSP page to be accessed when you invoke **Debug > Go**. The debugger connects to the Caché server process that is servicing the CSP page's HTTP request. Use this option for debugging CSP applications, for example, to step through the code for the Test.csp page, enter /csp/user/Test.csp as a debug target.

17.2.2 Breakpoints

A project maintains a list of breakpoints that you set with F9. When you start debugging a project's debug target (with **Debug > Go**), the breakpoints defined by the project are set in the target process.

To view breakpoints, select **Debug > Breakpoints > View Breakpoints**. To add and remove breakpoints, place the cursor at the breakpoint location and select **Debug > Breakpoints > Toggle Breakpoint** or F9. You can also add or remove breakpoints using **Project > Settings > Debugging > Breakpoints**.

Note: 20 is the maximum number of breakpoints that can exist in a routine. If more than 20 breakpoints are set, the Debugger displays `<ROUTINELOAD>^%Debugger.System.1` and halts debugging.

17.3 Debug Menu

Debug menu options are described below:

Attach	Displays a list of processes currently running on the Caché server and lets you attach to one to debug. If you select a process and select **OK**, Studio breaks into the selected target process and allows you to start debugging it. If you generated source for the current routine executing in the target process, the source is displayed in an editor window. If you later terminate debugging with **Debug > Stop**, the target process resumes executing.
Go	If you are not currently debugging, **Go** starts the target specified by the Project's debug target. (f you haven't set a target, you are asked for one. A debug target is the name of routine or method to execute; you can set this using the**Debug Target** dialog. Once the target is started, it runs until the first breakpoint. If you did not set any breakpoints in your application, it runs to completion without stopping.
Restart	Halts execution of the target process, restarts it, and resumes debugging (as if the **Go** command was used).
Stop	Stops debugging and either halts the target process or detaches from it. If the target process was running and attached to with **Attach**, then the target process continues running. If the target process was started as a result of the **Go** command, then it is terminated.
Break	Pauses execution of the target process (that is, if the debugger is attached to a target process that is currently running, not stopped).
Interrupt	Interrupts execution of the current command.

Step Into	Executes the current command in the target process and stops on the next command, *stepping into* any function calls or loop bodies.
Step Over	Executes the current command in the target process and stops on the next command. The debugger *steps over* any function calls or code blocks (such as loops) it encounters; it stops on the command following the function call or code block.
Step Out	Advances the execution of the target process by leaving or *stepping out* of the current code block or function and stops on the next command at this outer level.
Run To Cursor	Available only for documents containing INT routines. Starts execution of the target process and stops when it reaches the line on which the cursor is currently located. This is equivalent to setting a breakpoint at the current line in the editor window, executing the **Go** command, and clearing the breakpoint when the program halts.
Watch	Toggles Watch window Display.
Breakpoints	**Toggle Breakpoints** Sets or clears a breakpoint on the current line in the current document. **View Breakpoints**: Opens the **Breakpoints** dialog with which you can list, add, and remove breakpoints.
Debug Target	Enter a debug target – a method, routine, Zen page, or CSP page. See also “Debug Target”

17.4 Watch Window

The Watch Window displays a table in which you can watch the values of variables and simple expressions. All variables and expressions listed in the Watch Window (called *watchpoints*) are evaluated after each debugger operation (such as **Step Over**) and their resulting values are displayed in the second column of the Watch Window. If the value of a variable or expression changes after a debugger operation, it is displayed in red. If a variable in the watch list is undefined when it is evaluated then the value is displayed as: <UNDEFINED>. Similarly, any expression whose result is an error displays an error message for its value. Note that you can also see the value of a variable displayed in a hint in the debugger by hovering your mouse over the variable.

To add a variable or expression to the Watch Window, double-click an empty cell in the first column and enter the variable or expression. Alternatively you can use your mouse to highlight text in an editor window, drag it over an empty cell in the Watch Window and drop it. You can edit the contents of the Watch Window by double-clicking on a variable or expression and typing.

The following are examples of variables and expressions you could enter into the Watch Window:

- a
- a + 10
- a(10,10)
- $L(a)

- person.Name

You can also change the value of a variable in the target process by entering a new value in the Value column of the Watch Window.

17.4.1 Debugger Watch Window Context Menu

Right-clicking a debugger watch window displays the following context menu:

Remove	Removes the active variable from the watch list.
View As	Select view type from list.
Dump Object	Displays result of %SYSTEM.OBJ.Dump() on selected variable.
Refresh	Refreshes the watch list.
Remove All	Removes all active variables from the watch list.
Add to Watch	Adds selected element of array or object property to be added to watch menu as an independent entry.

18

Using Studio Templates

This chapter describes how to use Studio Templates.

Templates are a repeatable way to insert functionality into Studio editor windows. There are three types of templates:

- *Text Template*—(what is meant by the word *Template*) A *simple text template* inserts generated text into a document. An *interactive text template*includes user input. An example would be a Studio template inserts a block of standard HTML into a CSP file. To access text templates, select **Tools > Templates > Templates...**.
- *New Document Template*—Creates a new document window in Studio.

  New Document Templates constitute the list of new document types displayed by the **File > New** menu. To create a new document template, create either a simple or interactive text template with a *Mode* attribute of `new`.
- *Add-in Template*—Creates a new tool in Studio. An Add-in Template differs from a Text Template in that it does not inject text into a document and does not require an open document.

  Add-ins are available using **Tools > Add-ins**. For information on using the SOAP Web client wizard, see *Creating Web Services and Web Clients in Caché*. For information on using the XML Schema Wizard, see the appendix "Using the XML Schema Wizard" in *Using XML with Caché*.

Studio comes with a set of Caché-supplied standard Studio templates. In addition, you can create your own custom templates using CSP.

Note: To ensure that Studio templates open quickly, disable the automatic detection of proxy settings in Internet Explorer.

1. Open Internet Explorer and select **Tools > Internet Options** and the **Connections** tab.
2. Select **LAN Settings** and uncheck any checked boxes. Make sure nothing on this page is checked and then select **OK** twice to close the Internet Options dialog.

18.1 Accessing Studio Templates

You can open a template using **Tools > Templates**, as well as with the right-click menu in the editor window. You can open a template by selecting it from the list of recently-used templates (showing on the right-walk menu under the word **Templates...**) or from the Studio Template list, accessed with (**Tools > Templates > Templates**), which lists available templates. Each template is associated with one or more document types; only templates associated with the current window's document type are shown in the Template list.

There are two styles of template; simple and interactive. A *simple template* inserts text at the cursor point with no further user interaction. An *interactive template* displays one or more screens soliciting additional information, like a wizard.

Any text that is highlighted when you open a template is replaced by the template. Many templates use the currently highlighted text as input to the template program.

18.2 Caché-Supplied Standard Studio Templates

Studio comes with a set of templates. You can see a list of all system-supplied templates in Studio in the %SYS namespace, **Workspace** window, **Namespace** tab, under **CSP files**. To see templates usable in the current document, use **Tools > Templates > Templates**. These templates are described below.

Note: By default, Studio templates use a session timeout of 90 seconds. If you are entering data into a Studio template, the session ends after 90 seconds of no user input. For more information, see the section "Default Timeout" below.

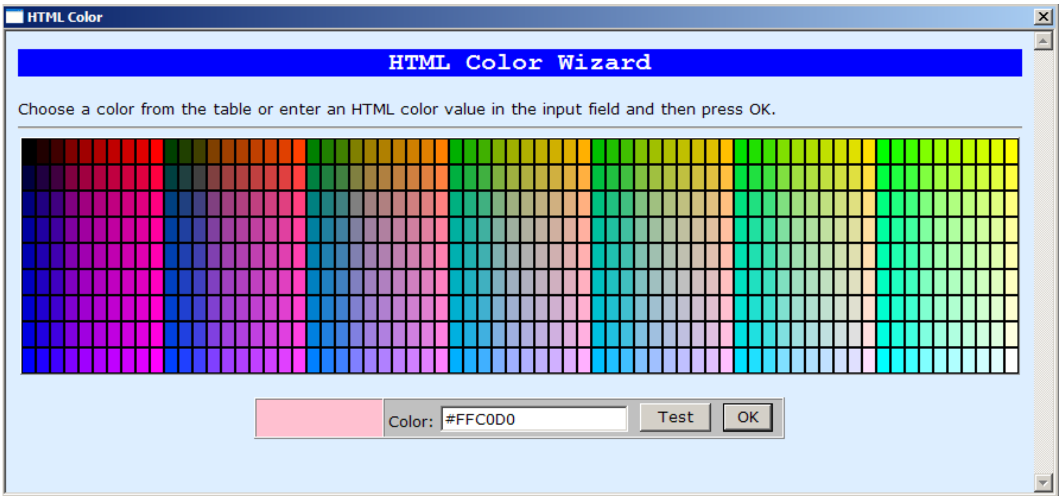
18.2.1 Templates

This section contains three tables defining the templates available in Studio for use with CSP or Zen:

Table 18–1: CSP Templates

Template	Description
HTML Color	Select to insert an HTML color value string (such as #F0F0F0) at the cursor point.
HTML Input	Select to insert an HTML input control at the cursor point.
HTML Script	Select to insert a <SCRIPT> tag at the cursor point, with the specified language and content.
HTML Table	Select to insert an HTML table at the cursor point with the specified characteristics. Select **Preview** to display a preview window.
HTML Tag	Select to insert an HTML tag at the cursor point, selected from a list with specified attributes. Or if you highlight an existing HTML tag and then invoke the template, you can edit the displayed attribute values.

Figure 18–1: Example of an Interactive Template, the HTML Color Table



Choose a color from the table or enter an HTML color value in the input field and then press OK.

Color: #FFC0D0

18.2.2 Class Definition Templates

Many of the CSP templates are available for use in class definitions (they can be useful in &html<> blocks). In addition, the following templates are available:

Table 18–2: Class Definition Templates

Template	Description
SQL Statement	Select to insert code for a specified SQL Statement at the cursor point. Select **Preview** to see test results of the table (using data in the database) in a popup preview window. You can specify whether the template returns only the SQL text, an embedded SQL cursor based on the SQL text, or a %ResultSet object based on the SQL text.
Web Form Wizard	Select to open a Wizard with which you can create a CSP form, specifying class members and a table style for the form to use. (See Building a Simple Application with Studio for an example.

18.2.3 Zen Templates

You can use the following wizards (also called templates) in Zen classes. For detailed information on using these wizards, see Zen Wizards in *Using Zen* and the chapters referenced in the table below for select templates.

Table 18–3: Zen Templates

Template	Description
Zen Chart Template	Select to insert a Zen chart definition within an Xdata block of a ZenPage class, selecting type, style, and attributes. For more information see the chapter "Zen Charts" of *Using Zen Components*.
Zen Element Template	Select to insert a Zen element within an Xdata block of a ZenPage class. You can insert and edit built-in, custom, or composite Zen components or create a new Zen component. For more information, see the chapter "Zen Layout" in *Using Zen*.
Zen Method Template	Select to insert a method. The wizard lets you select a scope, either instance or class, a location where the method will execute, a name, and whether to add a try/catch error processing template. You can further edit it in the Studio editor. See the section "Adding Client-Side Methods" in this book for an example of using the Zen Method Template.
Zen Style Template	Select to insert a CSS style declaration within an Xdata Style block of a Zen class. A table displays the CSS style declarations defined by Zen components. You can select one and override it within your page by editing details. For more information, see the chapter "Zen Style" in *Using Zen*.
Zen TablePane Template	Select to insert a new Zen tablePane definition within an Xdata block of a ZenPage class. Select the source of the query or table for this tablePane and then adjust properties as desired. For more information, see the chapter "Zen Style" in *Using Zen*.

18.2.4 Add-In Templates

The **Tools > Add-Ins** menu contains a list of wizards with which you can add items to your project. The menu contains the following add-ins.

Table 18–4: Add-Ins

Add-In	Function	For More Information
.Net Gateway Wizard	Imports a DLL assembly file from .NET and create a set of corresponding classes.	*Using the Object Gateway for .NET*
Activate Wizard	Creates Caché classes which provide you with access to COM objects from within Caché.	The Caché Activate Wizard in *Using the Caché ActiveX Gateway*
Java Gateway Wizard	Imports a class file or a jar file from Java and creates a set of corresponding classes.	*Using the Java Gateway* in the Ensemble documentation set
SOAP Wizard	Reads a WSDL (Web Services Description Language) document and creates one or more Web client classes or Web service classes.	*Creating Web Services and Web Clients in Caché*
XML Schema wizard	Reads an XML schema and creates a set of corresponding classes.	"Using the XML Schema Wizard" in *Using XML with Caché*.
XSL Translate Wizard	Transforms an XML file using a specified XSL stylesheet.	"Performing XSLT Transformations" in *Using XML with Caché*

18.3 Making Your Own Studio Templates

You can create any of the following types of templates:

- Simple Text Templates insert text at the current cursor location.
- Interactive Text Templates provide an interactive dialog that request information and then insert text at the current cursor location.
- Add-in Templates provide tools for general use in Studio.
- New Document Templates appear in the Studio New Document dialog and create new documents.

Note: You should have some familiarity with CSP development before attempting to create Studio templates.

18.3.1 Template Architecture

Studio Templates are created and implemented using Caché Server Pages (CSP); each template is one or more server pages that run on a Caché server. When you invoke a template, Studio creates a window containing a browser and makes an HTTP request (via the built-in Caché simple HTTP server) to the CSP page associated with the template. The CSP page can either:

1. Return HTML containing an interactive form that solicits additional input from the user, or

2. Return XML containing the text to be inserted at the cursor point in Studio. (All Text and New Document Templates do this as their final step; Simple Templates, with no user interface, implement only this step; Add-in Templates do not insert text into a document.)

To make it easier to develop templates, Caché includes a set of custom CSP tags that perform all the basic operations of templates. These tags are described in the following sections.

The power of templates comes from the fact that they are running on a Caché server and have the entire power of Caché at their disposal. Therefore, for example, templates can perform sophisticated database operations.

When invoking a template, Studio passes parameters to the server, where they are accessible, in the *%request* object. These parameters (which are case-sensitive) may include:

- *Project*—name of the current Studio project.
- *Namespace*—name of the namespace Studio is connected to.
- *User*—name of the current Studio user.
- *Language* —ObjectScript language mode.
- *Document Namespace*—namespace the document belongs to (might be different from current namespace.
- *Name*—name of the current document, if any.
- *Tabsize*—Number of spaces a tab contains in the document.
- *SelectedText*—selected text in the current document, if any.

Give each template a unique name (unless you are replacing an existing template with a new one).

18.3.2 Default Template Timeout

A session timeout is the amount of time in which a session stays open without any input from a user. At the end of this time of no user input, CSP closes the session.

Templates created in Studio are stored in either of the CSP applications, /isc/studio/usertemplates (which appear on the **Tools > Templates** menu) or /isc/studio/templates (which appear on the **Tools > Templates > Templates** menu). By default, both of these applications have default session timeouts of 90 seconds. In the Management Portal, on the CSP Application options page for each of these applications, you can enter a number in the Default Timeout setting, but the number has no effect on the hard-coded session timeout of 90 seconds. This is intentional to prevent inactive Studio templates from retaining system licenses.

18.3.3 Simple Text Templates

A simple template is a single CSP page that returns a block of text that is inserted into the current document at the cursor point. The CSP page contains the special Studio CSP tag: <csp:StudioSimpleTemplate>.

18.3.3.1 Creating a Simple Studio Template

To create a simple Studio template, start Studio and do the following:

1. Create a new CSP page with **File > New > Caché Server Page**.
2. Replace the contents of the document with the following. (The name of the new template is `MyTemplate`.)

```
<csp:StudioSimpleTemplate name="MyTemplate" type="CSP">
SOME TEXT!
```

3. Save and compile this CSP document with **Build > Compile**. It does not matter what file name you use for the CSP document or which namespace or CSP application you store it in.

   To make a Studio template accessible to all namespaces, save it in the %SYS namespace in /isc/studio/usertemplates.

You have now defined a template called `MyTemplate` (its name is specified by the *name* attribute of the <csp:StudioSimpleTemplate> tag). When you are in an open CSP page in the Studio Editor, select **Tools > Templates > MyTemplate**, and the text `SOME TEXT!` is inserted into the page.

The Template dialog lists templates whose type matches the document type from which you invoked the Template dialog. In our example, the `MyTemplate` template has a type of `CSP`. When you are in a CSP page in the Studio Editor, it appears on the **Tools > Templates** list.

You can also edit the type attribute, a comma-separated list of document types, to specify the document types the template can be called from. The list of types includes:

Table 18–5: Studio Template Types

Template Type	Template is available when your current document is a
CSP	CSP document
CSR	CSR document
MAC	MAC routine
INT	INT routine
INC	INC file
BAS	BASIC script
CLS	Class Definition document
MVB	MVBasic Routine

18.3.3.2 Testing a Simple Studio Template

To test a simple text template:

1. Open an existing (or create a new) CSP page in Studio (you can use any CSP application directory). While you can continue to use the namespace in which you created the template, you probably want to connect to the namespace where you normally work.
2. Position the cursor in the CSP document editor window and select **Tools > Template > Templates**.
3. You see your new template in the list. Select it and select **OK**.

The body of the simple template is inserted at the cursor location. For example:

```
<html>
<body>
SOME TEXT!
</body>
</html>
```

18.3.3.3 Adding Logic to a Simple Studio Template

The real power of templates is that they can execute code. CSP page templates can include CSP features including:

- Embedded expressions using `#( )#` syntax.
- Line of code contained in a `<script runat=”server”>` tag.

For example, we could create a simple template that returns the current date and time (using the **$ZDT** function) in an HTML `<B>` (boldface) tag:

```
<csp:StudioSimpleTemplate name="Now" type="CSP">
<B>#($ZDT($H,3))#</B>
```

18.3.3.4 Troubleshooting a Simple Studio Template

If you have problems developing a custom template, one way to debug it is to view your template CSP page in a browser with **View > Web Page** while editing your CSP. Use a browser capable of displaying XML text; the value returned by a template is wrapped in XML.

For example, you can display the output of the template defined earlier in this section by entering a URL in a browser (or using **File > Open URL**), such as:

```
http://localhost:8972/csp/user/MyTemplate.csp
```

This results in the following XML response:

```
<?xml version="1.0"?>
<template>
<![CDATA[BODY##www.intersystems.com:template_delimiter##
 SOME TEXT!##www.intersystems.com:template_delimiter##]]>
</template>
```

The response is contained in a `<template>` element. The body of the response is delimited using the `##www.intersystems.com:template_delimiter##` delimiter.

18.3.4 Interactive Studio Templates

An interactive template is a set of one or more CSP files that display a dialog window (using HTML) requesting user input. The interactive template's final page inserts a block of text into the current document at the cursor point, as a simple template does.

When you select an interactive template, Studio displays a dialog window containing a Web browser. The first page (or pages) of an interactive template returns an HTML form that solicits user input.

The first page of an interactive template starts with the <csp:StudioInteractiveTemplate> tag. The attributes of this tag are identical to those of the simple template tag. The final page (the one returning output to Studio) starts with the <csp:StudioGenerateTemplate> tag. Any intermediate pages (perhaps you are creating a multiple page wizard) do not need any special tags. The penultimate page contains a **SUBMIT** button that invokes the final page.

For example, suppose you want to create an interactive template, `MyScript`, that solicits script content and script type from a user and inserts a `<script>` tag into a CSP document. The first page, MyScript.csp, contains a simple HTML form:

```
<csp:StudioInteractiveTemplate name="MyScript" type="CSP">
<HTML>
<BODY>
<FORM NAME="form" ACTION="MyScript2.csp">
Language:
<SELECT NAME="language">
<OPTION VALUE="CACHE" SELECTED>CACHE
<OPTION VALUE="JavaScript">JavaScript
</SELECT>
<BR>
Script: <TEXTAREA NAME="script" ROWS="10" COLS="40"></TEXTAREA>
<BR>
<INPUT TYPE="submit" VALUE="OK" NAME="submit">
</FORM>
</BODY>
</HTML>
```

This CSP page contains:

1. A <csp:StudioInteractiveTemplate> tag specifying that this is an interactive template as well as the template's name and type.

2. An HTML form whose `ACTION` attribute links it to the final page of the template, MyScript2.csp (see below).
3. A SELECT control for specifying a script language.
4. A TEXTAREA control for entering the contents of the script.
5. A SUBMIT button that submits the contents of the form to the MyScript2.csp page.

The final template page, MyScript2.csp, uses the values submitted from the form to create a response that is sent back to Studio:

```
<csp:StudioGenerateTemplate>
<SCRIPT LANGUAGE="CACHE" RUNAT="SERVER">
    Write "<SCRIPT"
    Write " LANGUAGE=""",$G(%request.Data("language",1)),""""
    If ($G(%request.Data("language",1))="CACHE") {
        Write " RUNAT=""SERVER"""
    }
    Write ">",!
    Write $G(%request.Data("script",1)),!
    Write "<","/SCRIPT>",!
</SCRIPT>
```

This page starts with a <csp:StudioGenerateTemplate> tag and then includes ObjectScript code that sends back a `SCRIPT` tag (with appropriate attributes) to Studio. To test this, create a new CSP page and select **Tools > Templates > MyScript**. Enter some ObjectScript in the script box and select **Ok**. Script tags with your script are entered automatically into your new CSP page.

18.3.5 New Document Studio Templates

You can create a new document template by adding a `mode` attribute, with value of `new,` to either the <csp:StudioSimpleTemplate> or <csp:StudioInteractiveTemplate> tags.

Any template whose `mode` is `new,` appears, by name, in the New Document dialog (invoked by **File > New**). The results of the template are written into a newly created document.

For example, to create a new document template that creates new CSP pages, make a CSP file similar to this:

```
<csp:StudioSimpleTemplate name="MyCSPPage" type="CSP" mode="new">
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="FFDDFF">
<H1>New CSP Page</H1>
</BODY>
</HTML>
```

After saving this CSP file (you can save it under any CSP application) and compiling it, it appears in the Studio New dialog as `MyCSPPage` When selected, a new, untitled CSP document is created with the following contents:

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="FFDDFF">
<H1>New CSP Page</H1>
</BODY>
</HTML>
```

You can create more sophisticated New Document Templates by adding logic or by using an interactive template, as described above for Text Templates.

Parameters passed to the server in the *%request* object for a new document template are:

- *Project*—name of the current Studio project.
- *Namespace*—name of the namespace Studio is connected to.
- *User*—name of the current Studio user.

- *Tabsize*—Number of spaces a tab contains in the document.

18.3.5.1 New Class Definition Templates

If you are creating a New Document Template that creates a new class definition, you must perform an extra step in your template code: you tell Studio of the name of the new class so that it can create the correct internal data structures for the new class definition. This information is returned from the Template to Studio via data in the `%session` object's Data property stored under the subscripts `Template` and `CLASS`. At some point, your template should contain code similar to this:

```
<SCRIPT LANGUAGE="CACHE" RUNAT="SERVER">
    Set %session.Data("Template","CLASS") = classname
</SCRIPT>
```

18.3.6 Add Text to End of a Document

You can add text to end of a document template by setting an `InsertAtEnd` flag on the last page of either a Simple Template or an Interactive Template. If this flag evaluates to true (1), then studio inserts the template-generated text at the end of document for a non-class document or before the first `<Storage>` tag for a class document.

```
<SCRIPT LANGUAGE="CACHE" RUNAT="SERVER">
    Set %session.Data("Template","InsertAtEnd") = 1
</SCRIPT>
```

18.3.7 Add-in Studio Templates

You can create an add-in template by adding a `mode` attribute, with value of `addin,` to either the <csp:StudioSimpleTemplate> or <csp:StudioInteractiveTemplate> tags.

Any template whose `mode` is `addin,` appears, by name, in the Add-in dialog (invoked by the **Tools > Add-in**).

You can create Add-in Templates in the same manner as described above for Text Templates.

When invoking an add-in, Studio passes parameters to the server, where they are accessible, in the *%request* object. These parameters (which are case-sensitive) include the following:

- *Project*—name of the current Studio project.
- *Namespace*—name of the namespace Studio is connected to.
- *User*—name of the current Studio user.
- *Document*—name of the current document, if any.
- *SelectedText*—selected text in the current document, if any.

18.3.7.1 Adding Items to a Project

From an Add-in Template, you can instruct Studio to add one or more items to the current project by using the inherited method, **AddToProject**, on the final page of the Add-in Template.

For example, the following adds a class, MyApp.MyClass, to the current project:

```
<SCRIPT LANGUAGE="CACHE" RUNAT="SERVER">
    Do ..AddToProject("MyApp.MyClass.CLS") // note .CLS extension
</SCRIPT>
```

Note that when adding an item to a project in this way, you must append the type of item (.CLS, .CSP, .MAC, and so on) to the item name. Also note that the item must exist before you add it to a project; adding a class to a project does not automatically create such a class (but, perhaps, your Add-in Template does this using the %Dictionary class provided in the class library).

19

Studio Menu Reference

This chapter describes menu and keyboard options available from the Studio menus. The menus are:

- File
- Edit
- View
- Project
- Class
- Build
- Debug
- Tools
- Utilities
- Window
- Help
- Context Menus
- Studio Editor Keyboard Accelerators

19.1 File Menu

The **File** menu contains options for opening and saving documents and projects. Options vary depending on whether a file is open. See also the section "Keyboard Accelerators."

New Studio	Use to connect to a different Caché server.
Change Namespace ...	Use to change to a different namespace.

New ...	Use to create a new document (such as class definition, routine, or CSP file) in an editor window. You can also drag and drop files into Studio. Document types are grouped under four tabs: • **General** - for creating a new ObjectScript routine, Caché Basic routine, Caché class definition (with the New Class wizard), Caché MultiValue routine, Web Service/Client Configuration, or Web Service. • **CSP File** - for creating a new Caché Server Page, XML file, JavaScript file, or cascading style sheet (CSS) file • **Zen**— for creating a new Zen application, component, page, report, form, or Web Service. (For details, see Zen Wizards in *Using Zen*. • **Custom** - for a new DeepSee KPI. For information on DeepSee KPIs, see the *Advanced DeepSee Modeling Guide*.
Open	Opens a Allows to you open an existing item from the current Caché namespace and server. The Open dialog is displayed for the current namespace. Select a file type as needed. Select **Ctrl+A** to Select All. If the item is in use by another user, you can open it for Read Only access. To open automatically-saved backup files, see Save Automatically Backs Up Routines, Include, and CSP Files Studio allows you to edit class definitions, routines, and CSP files only from the current server and namespace. To open an item from a different server or namespace, use **File > Change Namespace** to change to a different namespace or switch to a different server. Use **File > New Studio** to open a second Studio window. If you select an item and right-click **Delete**, the item and all subitems are deleted. Examples: if you select an .INT file, both .INT and .OBJ files are deleted. If you select a .cls file, all associated .INT and .OBJ files are deleted also.
Open URL	Displays HTML source in an editor. This is useful when you are developing a Web-based application to view progress.
Close	Closes the current editor.
Save	Saves the contents of the current editor.
Save As ...	Saves the contents of the current editor with a name that you specify.
Save All	Saves the contents of all open windows.
New Project	Creates a new project in the current Caché server and namespace.

Open Project ...	Opens an existing project in the current Caché server and namespace. You can also drag and drop a project into Studio. To open a project from a different server or namespace, use **File > Change Namespace** to change to a different namespace or switch to a different server. Use **File > New Studio** to open a project in a second Studio window.
Save Project	Saves setting for the current project. It does not save any modified documents belonging to the project.
Save Project As	Saves the current project with a name you specify.
Close Project	Closes the current project.
Print ...	Prints the current document.
Print Preview	Displays the document as it will look when printed.
Print Setup ...	Opens the Print Setup dialog with which you can set how documents are printed.
Recent Files ...	Lists recently used files. **More** shows files (if they exist) in categories **Today**, **Yesterday**, **Last 7 days**, **Last 30 days** and **All**. **All** is limited to 100 documents. Select **Clear History** to clear all. Select **Open Document** to open a selected document.
Recent Projects ...	Lists recently used projects.
Exit	Ends the current Studio session.

19.2 Edit Menu

The **Edit** menu contains editing and navigation options. Most of the options have keyboard shortcuts; see the section "Keyboard Accelerators.".

19.2.1 Basic Editing

Undo	Reverses the last action. Note that changes made to classes with the Class Inspector cannot be reversed using **Undo**.
Redo	Reverses the most recent **Undo**.
Cut	Deletes the current text selection and copies it to the clipboard.
Copy	Copies the current text selection to the clipboard.
Paste	Inserts the contents of the clipboard at the current cursor location.
Delete	Deletes the current text selection without copying it to the clipboard. If you highlight an item in the Workspace window, the item and any of its generated files are deleted.
Select All	Selects all the contents of the current document.

19.2.2 Find and Replace

Find	Searches for text in the current document. You can use wildcard matching with the * (asterisk to match any number of any character) and ? (question mark to match a single character). To find the character * or ? or \ (asterisk, question mark, or backslash) , escape it with a backslash (\). To find a tab, use `\t`. See the section More on Find beneath this table to specify an element type to search within and an explanation of the backslash escape.
Find In Files	Searches for text in multiple files on the Caché server. Enter a string to search for, select the type of file to search (such as .cls for class definitions), and select **Find**. See the section Find in Files beneath this table for specifics.
Search	Searches for a class in the current project. In the Search window, type in a class name. The list shows matching entries. To open a class, double-click a class or select a row and press Enter or select **GoTo**.
Cancel	Cancels a running Find in Files search or a class compilation.
Replace	Replaces one text string with another in the current document.
Go To	Moves the cursor to a location in the current document, specified as either a line number or (for routines and class definitions) a tag or class member.
Go Back	After a **Go To** action, returns the cursor to the previous location — before the **Go To** action.
Bookmarks	See "Bookmarks" below.
Advanced	See "Advanced"

Next Error	Moves the cursor to the next error in a CSP file.
Previous Error	Moves the cursor to the previous error in a CSP file.
Next Warning	Moves the cursor to the next warning in a file.
Previous Warning	Moves the cursor to the previous warning in a file.

19.2.2.1 More on Find

Backslash Escape

The search engine normally interprets a backslash (\) as a metacharacter; that is, a character that means something other than itself. In this case — the backslash and the following character form a two-character code. When you want to search for the backslash itself, you need to create a two-character code since the search engine always looks for a second character when it sees a backslash. Create a two-character code with a second backslash. The search engine interprets this as the backslash character itself.

This convention was implemented during the development of the UNIX grep command and the convention, if not the underlying C code, has been duplicated many times since.

Match Element Type

To find text that is in a particular element type (such as a command, variable, operator, and so on), enter the desired text in the **Find what** field and select the **Match Element Type** check box.

Note: **Find** displays the searched-for text in all elements of that type in the open file, *regardless of language selected*. In the **Language** field, select the language that you are interested in *to limit the number of element types shown* in the **Element** list. In the **Element** field, select the element type that contains the text you are looking for and select **Find Next**.

For example, searching for the word `Set` in a **Comment** with the **Class Definition Language** selected matches all instances of the word `Set` in comments that exist in any language in the file.

19.2.2.2 Find in Files

When you select **Find** in the Find in Files dialog, Studio searches the selected files in the current Caché namespace and returns a list of all (up to the first 5,000) files that contain the search string. Double-click an item in the search results to open the file and display the item, highlighted. Line & column numbers for the selected item are displayed in the right corner of the status bar.

Note that Find in Files searches stored data; it does not search modified open documents. If you search only in the current project and the current project is either a new project or a modified project, you are prompted to save the project. If you refuse, Find in Files is canceled.

Note also that to find a backslash (\) in Find in Files, you need to escape the backslash with another backslash. See the section More on Find for an explanation of the backslash escape.

The Filter field can contain the elements in the list below. You can use SQL AND and OR logical operators to enter more than one filter. For example, `Type=5 AND Modified>01/01/08`. The contents of the Filter field forms part of an SQL WHERE clause. The fields come from the %Studio.OpenDialogItems class.

You can enter your own custom mask in the **In files/file types** field, such as al*.mac. Use a comma delimited list to enter multiple filters in any field.

You can use the following items in the **Filter** field.

- `IsTrue=1` or `0` - True (1) if this is a document, false (0) if it is a directory.
- `Name=`*file name* - Enter a file name to search within selected files.

- `Characters=`*number of characters* - Filters for documents of a certain size. Can include SQL relational operators.
- `Type=#` - Type is followed by an integer which filters according to file type list, shown in %Studio.OpenDialogItems.Type. This can filter to a finer degree than the file type field. To search for only .mac files, for example, enter `Type=5`.
- `Modified=`*last modified timestamp* - Can include SQL relational operators.
- `Generated=1` or `0` - True (1) if this is a generated document, false (0) if it is a user-created document.
- `Description=`*description* - Enter a description to search for within files.

19.2.3 Bookmarks

You can use bookmarks to keep track of locations in your application source. Bookmarks are stored on the local machine in which Studio is running; they are not shared among multiple users. The bookmark options are:

Toggle Bookmark	Adds or removes a bookmark at the current line in the current document.
Clear Bookmarks	Removes all bookmarks defined for the current document.
Next Bookmark	Moves the cursor to the next bookmark in the current document.
Previous Bookmark	Moves the cursor to the previous bookmark in the current document.

19.2.4 Advanced Editing

The Advanced Editing menu contains some commands that are displayed in certain circumstances only.

Expand Commands	Displays when you have an ObjectScript routine open and text is highlighted. Replaces all abbreviated ObjectScript commands contained in the currently selected text with their full names. For example, the following code:
	`S x = 10`
	Is replaced with:
	`Set x = 10`

Compress Commands	Displays when you have an ObjectScript routine open and text is highlighted. Replaces all ObjectScript commands contained in the currently selected text with their abbreviated versions. For example:
	`Set x = 10`
	Would be replaced with:
	`S x = 10`

Increase Line Indent	Increases indent for selected lines.
Decrease Line Indent	Decreases indent for selected lines.

Make Uppercase	Uppercases selected text.
Make Lowercase	Lowercases selected text.
Comment Line	Turns a selected line into a comment by adding a # (pound sign) to the beginning of the line.
Uncomment Line	Turns a selected comment into a regular line by removing a # (pound sign) from the beginning of the line.
Comment Block	Turns a selected block of text into a commented block by adding a /* (slash asterisk) to the beginning of the block and a */ (asterisk slash) to the end of the block..
Uncomment Block	Turns a commented block of text into a regular block by removing the /* (slash asterisk) from the beginning of the block and the */ (asterisk slash) from the end of the block..
Tabify Selected Lines	Adds a tab to the beginning of each of a set of selected lines.
Untabify Selected Lined	Removes a tab from the beginning of each of a set of selected lines.

19.3 View Menu

The **View** menu contains options that control what is displayed. Which options are shown on this menu depends on where your cursor is. See also the section "Keyboard Accelerators."

Workspace	Shows or hides the Workspace window. The Workspace window has three tabs: • Project - Displays the contents of the current project. • Windows - Displays a list of all current windows. • Namespace - Displays the contents of the current namespace.
Inspector	Shows or hides the Inspector. The Inspector displays class definitions in an editable table. Some aspects of class definitions can be changed in a file only; some can be changed in a file or in the Inspector; and some can be edited only in the Inspector.
Output	Shows or hides the Output window. The Output window displays output from the server (such as error messages resulting from compilation). You can enter ObjectScript commands into this window; they are executed on the server. Line & column numbers for the selected information are displayed in the right corner of the status bar.
Watch	Shows or hides the Watch window. The Watch window displays watchpoints when debugging.
Toolbars	Lets you select Studio toolbars to show or hide.
Full Screen	Expands Studio to occupy the full screen.

Text Size	Lets you Increase, return to Normal, or Decrease text size of text in Studio.
Show Special Characters	Displays spaces, tabs, and end-of-line characters in the Studio Editor window.
Show Line Numbers	Displays line numbers.
Reload	If a document is active, reloads the saved version of current document. If the Workspace window is active, reloads the window or project. If the Workspace window, Namespace tab is active reloads the subtree parent of the selected item; highlight topmost level to reload the entire tree.
View Other Code	Displays any source code generated by the compiler, such as .INT and .MAC files, related to the position of the cursor. This option works only if the current window has one or more source files currently generated for it.
View Other Documents	Displays other documents related to the current document. In some situations,**View Other Code** results in the same behavior as **View Other Documents**.
Web Page	Available only when in a CSP file. Opens your default Web browser and displays the current CSP file, so that you can see how your CSP pages will look as you develop Web applications. You can change the network address portion of the URL used with **Project > Settings**. The default value is `http://localhost:8972`.
Expand Code	Available only in some files. Displays complete code in text editor window.
Contract Code	Available only in some files. Contracts some code sections in text editor window. Select the plus icon to expand a section.
Show Class Documentation	Available only in class definition files or when a class is selected. Displays online documentation for the current class derived from the (saved) descriptions of class members.
Language Mode	Available only when you are in a source code file, such as .INT or .MAC. Select from a list of Caché language versions as appropriate for your site.

19.3.1 Toolbars

The Toolbars menu lets you toggle the display of toolbars and lets you customize toolbars.

Standard	Toggles display of the Standard toolbar.
Debug	Toggles display of the Debug toolbar.
Members	Toggles display of the class Members toolbar.
Status Bar	Toggles display of the status bar at the bottom of the Studio window. From left to right, this bar shows a status message and the location of the cursor by line and column. The four buttons on the right, if highlighted, show that the **Caps Lock** key is on, that the **Num Lock** key is on, that the **Insert** key is on (Overwrite), and that the current file is a read-only file. The status bar also displays line and column numbers for Find in Files and Output windows, where appropriate.
Bookmark	Toggles of the Bookmark toolbar.
Customize	Opens the Customize dialog.

Toolbars, displaying text labels are shown below. To display text labels in Studio, choose **View > Toolbars > Customize**, the Toolbars tab. Select a toolbar and select **Show text tabels**. (If a toolbar is already selected, uncheck the toolbar, recheck the toolbar, and check **Show text labels**.)

Figure 19–1: Standard Toolbar

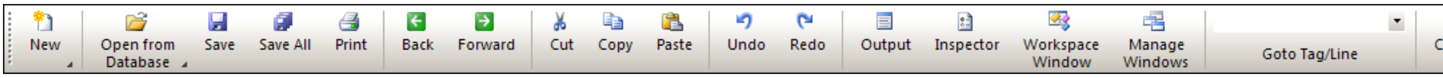


Figure 19–2: Debug Toolbar

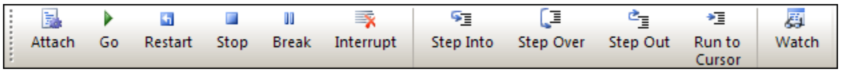
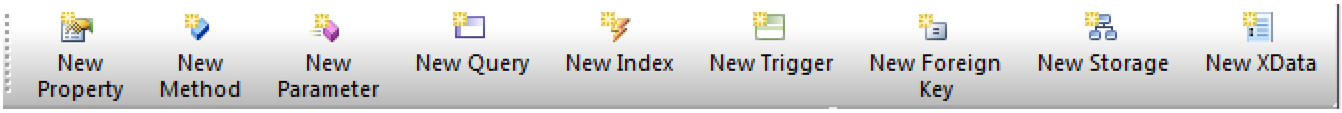


Figure 19–3: Class Members Toolbar



The BPL toolbar is only applicable in Ensemble.

Figure 19–4: BPL Toolbar

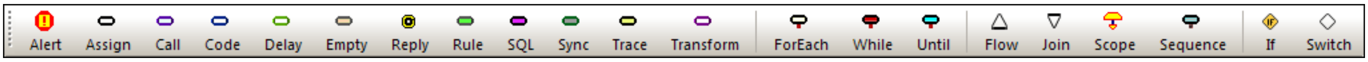
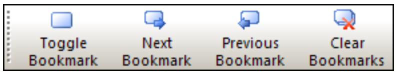


Figure 19–5: Bookmarks Toolbar



19.3.2 Customize Toolbars

The Customize menu consists of four tabs for customizing parts of the Studio interface, Commands, Toolbars, Tools, and Options.

Commands	Lists menus and all commands on Studio menus. Drag a command and drop it into an open toolbar. To remove a command from a toolbar, drag it off the toolbar.
Toolbars	Select check boxes to display toolbars. Toolbars can include text labels if you select **Show text labels.** The Menu Bar cannot be cleared. Select **New** to create a new toolbar.
Tools	Adds menu items to the Studio Tools menu. Specify an item name, it's command, any arguments, and it's initial directory.
Options	Select check boxes to turn on the display of screen tips, screen tips including shortcut keys, and large icons.

19.4 Project Menu

The **Project** menu contains options for working with projects.

Add Item	Adds the item in the current editor window to the current project.
Remove Item	Removes the item in the current editor window from the current project.
Settings	Select to edit settings for the current project: • **General**: *HTTP Address used by Studio to set Web pages for this project*: Set URL location for Web pages for this project. **Defines**: Define a macro that is applied when you compile the project. You can define a debug macro which can be tested by other macros and turns on additional checking in the code. For example, **debug=1** defines the macro $$$debug to be 1. • **Debugging**: Set **Debug target** and **Breakpoints**. See the chapter "Using the Studio Debugger" for details.

19.4.1 Common Project Tasks

To delete a project, select **File > Open Project** and right-click the project that you want to delete.

To import a project, select **Tools > Import Local** and select the .xml file that contains the project.

To export a project, open the project, select **Tools > Export**, select **Export Current Project**, browse to the output directory, and enter a file name.

Project names cannot include the characters .,;/\ (that is, period, comma, semi-colon, slash, or backslash).

19.5 Class Menu

The **Class** menu contains options for editing class definitions and is available only when you are in a class definition file.

The class options are arranged in an **Add** submenu and an **Override** dialog. The options include:

Add	Select one of: **Property**, **Method**, **Class Parameter**, **Query**, **Index**, **SQL Trigger**, **Foreign Key**, **Storage**, **Projection**, or **XData** Opens a wizard for the item and inserts an item definition into the current class definition. See separate chapters in this book for details on options for each of these class members.
Refactor or Override	(Refactoring is available only when Studio is connected to a Windows server (it may be Override on other platforms). Some features may partially work on non-Windows platforms, but you should not use these features, because you may get unexpected results.) Override: Opens a window that lists items that the current class inherits that you can select. It inserts an override definition into the selected class definition. Items listed include **Properties**, **Methods**, **Parameters**, **Queries**, **SQL Triggers**, and **XData** routines. Rename: Enter a new name. The new name replaces the old name in all locations in the document. (Studio does not refactor code inside literal strings, such as in an embedded SQL statement or in a method that takes a string with a column name.) . If you check **Reset Storage** (for a class) or **New Storage Slot** (for a property), the renamed item is created without storage and default storage is generated. You see a confirmation box which If you choose to delete an old class, its storage extent definition is also deleted. Refactoring delays applying changes to the database until you are finished reviewing changes across all documents. When you select **Accept Changes**, changes are saved in a temporary location. When you select **Finish** all changes are applied. If you select **Finish** but haven't accepted changes to all documents, you are prompted `Not all documents modified. Proceed anyway?` and you can accept changes for more documents or finish. Note: Do not use refactoring in a production environment. Use only during development. Studio does not allow any data manipulation.
Superclasses	Lists superclasses of the current class alphabetically. You can pick from these to **Add to Project**, **Show Documentation**, or **Open**.
Derived Classes	Lists classes derived from the current class (subclasses). You can pick from these to **Add to Project**, **Show Documentation**, or **Open**.
Create Subclass	Displays the New Class Wizard. A class created using this option becomes a derived class of the class in the current window and inherits its members, including properties, methods, class parameters, applicable class keywords, and the parameters and keywords of the inherited properties and inherited methods.

A projection automatically creates related files for other languages when you compile a class. For example, adding a Projection of type %Projection.Java generates a new Java class when it compiles so that you can use your class from a Java application.

19.6 Build Menu

The **Build** menu contains options for compiling and building applications. The behavior of the **Build** options is controlled by the Compile settings in the Studio Options dialog.

The build options include:

Compile	Compiles the contents of the current window. Uses settings of the **Compiler** tab from **Tools > Options**. Any messages from the compiler are displayed in the Output Window. If **Skip Related Up-to-date Classes** is enabled then: • The current class definition is only compiled if it has been modified. • If possible, Studio performs an incremental compile; if the only change to a class definition is in the *implementation* of one or more methods then only these methods are compiled.
Compile with Options	Use to select options for this session only or to change the default compile options. Default options can also be set with the **Tools > Options, Compiler** tab
Rebuild All	Compiles all the components in the current project whether or not they have been modified from the last compile.

19.7 Debug Menu

The **Debug** menu contains debugging options. To see the Debug Menu options, see the section “Debug Menu” in the chapter “Using the Studio Debugging .” See also the section “Keyboard Accelerators.”

19.8 Tools Menu

The **Tools** menu contains miscellaneous options.

The tools options include:

Class Browser	Opens the Studio Class Browser. The Class Browser displays a list of all classes in the current namespace as well as their members (defined and inherited).
Show Plan for SQL Statement	Opens dialog for an SQL statement. An SQL statement selected in the active document is displayed and editable in the dialog. Selecting **Show Plan** displays the query execution plan in a web page.
Templates	Displays a list of Studio Templates. A template injects a stream of text at the current cursor location. Studio provides templates. In addition, you can create your own. For more information see the chapter on “Studio Templates”.
Add-Ins	Contains a list of wizards that help you add items to an open Studio file or connect to existing files using standard formats. The wizards are the .NET Gateway Wizard, the Activate Wizard, the Java Gateway Wizard, the SOAP Wizard, the XML Schema Wizard, and the XSL Transform Wizard. For more on these wizards, see the section “Add-In Templates”.

Task List	Displays list of tasks. You can add, edit, or delete a task in the New Task window. Each task includes a server, namespace, document name, line #, and optional description. The current line code or selected text is used by default for the task description. **GoTo** takes you to selected document and line #. If needed, Studio connects to specified by task server/namespace using current security credentials.
Export	Exports one or more items (class definitions, projects, routines, include files) to either a local file or a file on the server system.
Export Special	Export RO, the format created with the %RO utility.
Import Remote	Lets you import an item from a remote file (that is; a file that is on the same machine as the server that Studio is connected to). All imported items are placed into new document windows. You can use this option to import a project, class definition, routines, or include files from either XML, or .RTN (%RO Routine format files). The **Import Remote** option displays a dialog that lists all the items contained in the file from which you can select. You can also specify whether the imported items should be added to the current project and if they should be compiled. The hand icon ✋ indicates that the file you are importing is older than the file on the system.
Import Local	Lets you import an item from a local file (that is a file on the same machine as Studio). All imported items are placed into new document windows. You can use this option to import a project, class definition, routines, or include files from either XML, or .RTN (%RO Routine format files). The **Import Local** option displays a dialog that lists all the items contained in the file. You can select which items you want to import. You can also specify whether the imported items should be added to the current project and if they should be compiled. The list of items to import must be less than 32K. The hand icon ✋ indicates that the file you are importing is older than the file on the system.
Compare	Compares an open file to one that you select with **Browse**. You must have specified an external compare tool with the **Compare** setting in **Options > Environment > General**. To work correctly, the compare tool must be able to accept command line parameters as **tool.exe** *file1 file2*. Tested compare tools are Microsoft Windiff and Perforce p4Diff.exe.
Copy Class	Creates a copy of an existing class with a new name.
Generate C++ Projection	Creates C++-related files when this class is compiled so you can use this class from a C++ application. For more information, see the chapter Using the C++ Binding in the book *Using C++ with Caché*.
Import and Export Settings	Opens Import and Export Studio settings wizard. In this wizard you can set file and directory for import and export and save these settings to a text file in format compatible with regedit.exe. This file can be imported by wizard or executed directly from command prompt or explorer on any windows computer. Import and reset settings cause a Studio restart. Import on Windows Vista requires administrator privileges to run.

Options	Lets you set Studio options. For details on options see the chapter "Studio Options."
Customize	Opens Customize window, in which you can customize aspects of the Studio UI, such as menus and toolbars.

19.9 Utilities Menu

The Utilities menu contains links to resources outside of Studio, such as:

- Management Portal
- Telnet
- Ensemble Management (available only if Ensemble is installed)

19.10 Window Menu

The **Window** menu contains standard window options for manipulating the windows in Studio.

19.11 Help Menu

The **Help** menu contains options for accessing online Help.

The help options include:

Studio Documentation	Displays the table of contents for Studio documentation.
Studio Commands	Displays the list of Studio options and short descriptions.
Online Documentation	Displays the home page of the Caché Online Documentation.
ObjectScript Reference	Displays the ObjectScript online reference.
SQL Reference	Displays the Caché SQL online reference.
Class Definition Syntax	Displays the online reference for class definition syntax.
InterSystems on the Web	Provides links to useful pages on the InterSystems Web Site.
About Studio	Displays version information about Studio.

19.12 Context Menus

Right-clicking areas in Studio displays different context menu. These include those described in the following sections.

19.12.1 Editor Context Menu

Right-clicking in the Studio Editor window displays a context menu. Within this context menu are many items available from the main menus, such as on the Editor menu are Cut, Copy, Paste, Find, Toggle Breakpoint, and Go Back, and so on. There are also additional options that are not available from the main menu. These include:

Help	The **Help** option displays context-sensitive help for selected syntactical elements. To use, right-click text and select **Help**. For example, if you right-click the word **Do** in ObjectScript code and select **Help** , the reference page for the Do command is displayed in your browser.
Add Task	Opens the Add Task window, in which you can add a task to the task list. See also Tools Menu.
Set Syntax Color	Displays a dialog in which you can choose the color used to display a specific syntactical element. To use, right-click text and select **Set Syntax Color**.
Goto <TAG>	Available when editing an ObjectScript routine. Lets you jump to the code that defines the ObjectScript tag. To use, in an ObjectScript routine, right-click a tag and select **Goto <TAG>** . If the right-clicked tag is defined in another routine, Studio automatically opens this routine.
Set *current item* as debug target	Sets the current item as the debug target.
Toggle Word Highlight	Highlights all instances of the word the cursor is pointing to in the document .

19.12.2 Workspace Context Menu

Right-clicking the Workspace window displays a context menu. Which menu is displayed depends on the cursor location. Different context menus are displayed if the cursor is on a package, a class, a CSP file, and so on. This table below shows items on the Workspace Package context menu not available on the menu bar.

Add	Adds a class selected from the displayed list to the current package.
Remove Package “name”	Removes the current package from this project.
Compile Package “name”	Compiles the current package.
Delete Package “name”	Deletes the current package.
Export	Exports the current package to an xml file. (To import a package, select **Tools > Import** and select an xml file.)
Add to Project	Adds the highlighted item to the current project.

Source Control	Select from Source Control options **Check Out**, **Undo Checkout**, **Check In**, **Get Latest** For more information see the section on "Source Control Hooks".
Close	Closes the current document.
Close All But This	Closes all documents except the highlighted document.
Close All	Closes all documents.

19.12.3 Inspector Context Menu

Right-clicking the Inspector window displays a context menu with the following items (if they are applicable to the current document in the editor window):

Add	Adds a member selected from the displayed list to the current class definition.
Override	Opens a window that lists items inherited by the current class, from which you select. An override definition is inserted into the current class definition window.
Reset to Default	Resets selected item to default.
Delete	Deletes the displayed item.
Locate	Available when a class member is displayed. Displays the member in the editor window.
Show Inherited Members	Toggles the inclusion of inherited members in the window display.
Show Headers	Toggles the display of the column headers, Name and Value, in the Inspector window.

19.12.4 Tab Context Menu

Right-clicking the tabs header displays a context menu with these items:

Close	Closes the current tab.
Close All But This	Closes all tabs except the current tab.
Close All	Closes all tabs.

19.12.5 Window Display Context Menu

Right-clicking a window where no other context menus apply shows the generic window context menu:

Floating	Disconnects the selected window from a fixed location; that is, it can be dragged freely to a desired locaiton.
Docking	Glues the selected window to a default location.
Hide	Conceals the selected window.

19.12.6 Debugger Watch Context Menu

To see the Debugger Watch Window Context Menu, see the section Debugger Watch Context Menu in the chapter "Using the Studio Debugger" in this book.

19.13 Keyboard Accelerators

This section lists Studio's keyboard accelerators (keyboard shortcuts)—combinations of keys that, when pressed, perform a Studio function. These are listed in the following table.

block of text

> In the following table a *block of text* means a number of whole lines. To select a block of text, put the caret at the start of the first line, press **Shift**, and select the **down arrow** till all of the relevant lines are highlighted. The caret is then displayed at the start of the line beneath the last whole line.

Accelerator	Action
General	
F1	Context Help
F4	Change Namespace or Connection
F8	Toggles Full Screen Display of Studio menus and editor window.
Ctrl+N	New Document
Ctrl+O	Open Document
Ctrl+Shift+O	Open Project
Ctrl+P	Print Opens the Print dialog. If text is selected, `Selection` is checked.
Ctrl+S	Save
Ctrl+Shift+I	Export
Ctrl+I	Import Local
Display	
Ctrl++	Expand All Expands all sections in the document that can be expanded. Select minus icon to collapse a section or **Ctrl+-** to collapse all sections.

Accelerator	Action
Ctrl+Left Select plus icon	Expand All Block Sections Expands all sections in this code block that can be expanded. Select minus icon to collapse a block or **Ctrl+-** to collapse all blocks.
Ctrl+-	Collapse All Collapses all sections that can be collapsed.
Ctrl+W	Show Class Browser
Ctrl+Shift+V	View Other Opens documents related to the current document, such as MAC or INT routines.
Alt+1	Toggles Inspector window display
Alt+2	Toggles Output window display The Output window has tabs for Result and Find in Files.
Alt+3	Toggles Workspace window display
Alt+4	Toggles Watch window display
Alt+5	Toggles Code Snippets window display
Alt+6	Toggles Find in Files window display
Alt+7	Toggles Class View window display
Ctrl+Alt++	Increase Font (Press Ctrl and Alt and the equal sign key — here called the plus sign.)
Ctrl+Alt+-	Decrease Font (Press Ctrl and Alt and the minus key.)
Ctrl+Alt+Space	Toggles display of Whitespace Symbols, spaces, newlines, and tabs
Ctrl+B	Toggle Bracket Matching Turns bracket matching on and off for the current window.
Ctrl+Shift+N	Toggles Line Numbers Display.
Ctrl+Tab	Next Window
Ctrl+Shift+Tab	Previous Window
Navigation	
Home	Go To Beginning of Line Subsequent presses hops the caret between the beginning of the line and the beginning of text on the line.
Ctrl+Home	Go To Beginning of Document
End	Go To End of Line

Accelerator	Action
Ctrl+End	Go To End of Document
Ctrl+-	Back
Ctrl+Shift+-	Forward
Page Up	Page Up
Page Down	Page Down
Ctrl+Page Up	Go To Top of Visible Page
Ctrl+Page Down	Go To Bottom of Visible Page
Ctrl+	Scroll Down
Ctrl+	Scroll Up
Ctrl+G	Goto
Ctrl+Shift+G or **F12**	Goto Documentation for Tag
Ctrl+F3	Go To Next Error
Ctrl+Shift+F3	Go To Previous Error
Alt+F3	Go to Next Warning
Alt+Shift+F3	Go to Previous Warning
Ctrl+]	Go To Bracket Moves the cursor between the innermost pair of brackets (or parentheses or braces). Pairs of all three kinds (one of each) can be highlighted if they are nested. This accelerator works only if bracket matching is turned on (see **Ctrl+B**).
Editing	
Insert	Toggle Insert/Overwrite Mode Toggles between *Insert* mode (new characters are inserted when typing) and *Overwrite* mode (new characters replace existing characters when typing).
Ctrl+Delete	Delete Next Word or to End of Word If caret is at the start of a word, deletes the word. If caret is in the middle of a word, deletes from the caret to the end of word.
Ctrl+Backspace or **Ctrl+Shift+Delete**	Delete Previous Word or to Start of Word If the caret is at the end of a word, deletes the word. If caret is in the middle of a word, deletes from caret to start of word.
Ctrl+Shift+L	Delete Line
Ctrl+C or **Ctrl+Insert**	Copy
Shift+Delete or **Ctrl+X**	Cut
Ctrl+L	Cut Line
Ctrl+V or **Shift+Insert**	Paste

Accelerator	Action
Ctrl+A	Select All
Ctrl+Y or **Ctrl+Shift+Z**	Redo
Ctrl+Z	Undo
Ctrl+Space	Show Popup If the cursor is in an appropriate location, this displays the Studio Assist popup, which shows options available for this location (such as classes, methods, properties, and so on, as appropriate).
Ctrl+~	Toggle Tab Expansion Toggles whether tabs or spaces are entered when you press Tab.
Ctrl+U	Uppercase Selection
Ctrl+Shift+U	Lowercase Selection
Ctrl+Alt+O	Toggles the Delay Parsing option.
Ctrl+Alt+U	Titlecase (Initial Caps) Selection
Ctrl+(	Insert Open and Close Parentheses. (Does not work on German and Swiss keyboards.*)
Ctrl+{	Insert Open and Close Braces.
Ctrl+[	Insert Open and Close Square Brackets.
Ctrl+<	Inserts Open and Close Angle Brackets.
Ctrl+=	Indentation Cleanup. Cleans up indentation on a selected block of whole lines of text.
Ctrl+/	Comment Line Turns a selected line into a comment by adding a # (pound sign) to the beginning of the line.
Ctrl+Shift+/	Uncomment Line Turns a selected comment into a regular line by removing a # (pound sign) from the beginning of the line.
Ctrl+/	Comment Block of Text Pressing **Ctrl+/** while a block of text is selected comments the block of text; that is, adds #; (pound semi-colon) to the start of each line (for Objectscript routine) or appropriate marker based on the document's language. (Does not work on German and Swiss keyboards.*)
Ctrl+Shift+/	Uncomment Block of Text Pressing **Ctrl+Shift+/** while a block of text is selected uncomments the block of text (that is, removes the #; from the start of each line for Objectscript routine — or other marker based on the document's language). (Does not work on German and Swiss keyboards.*)

Accelerator	Action
Ctrl+Alt+/	Comment Markers Added to Block of Text Inserts block type comments (such as /*...*/) for specific language if block comments are supported. If block type comments do not exist, then single line comment marker is inserted. (Does not work on German and Swiss keyboards.*)
Ctrl+Shift+Alt+/	Comment Markers Removed from Block of Text Removes block type comments (such as /*...*/) for specific language if block comments are supported. If block type comments do not exist, then single line comment marker is inserted. (Does not work on German and Swiss keyboards.*)
Ctrl+E	In an ObjectScript document, commands in a selection are replaced with their full names.
Ctrl+Shift+E	Compress Commands In an ObjectScript document, commands in a selection are replaced with their abbreviated names.
Ctrl+.	Insert Dots With a block of text selected, Insert dots at the start of each line (after leading white space). Lines must start with leading whitespace. This is for use in block structuring with leading periods with argumentless **DO** commands.
Ctrl+Shift+.	Remove Dots Remove leading dots from the start of the selected block of text (for use in block structuring with leading periods with argumentless **DO** commands).
Ctrl+Shift+T	Add Task
Find and Replace	
Ctrl+F	Find
F3	Find Next
Shift+F3	Find Previous
Ctrl+Shift+F	Find in Files
Ctrl+, (comma)	Search
Ctrl+H	Replace
Ctrl+Shift+G	Go To
Ctrl+Alt+G	Go Back
Bookmarks	
Ctrl+F2	Toggle Bookmark on Current Line
F2	Go to Next Bookmark
Shift+F2	Go to Previous Bookmark
Ctrl+Shift+F2	Clear All Bookmarks

Accelerator	Action
Build and Compile	
F7	Rebuilds All Documents in Project
Ctrl+F7	Compile Active Document
Ctrl+Shift+F7	Compile with Options
F5	View as Web Page
Debugging	
Ctrl+Alt+L	Toggle Studio Debug Logging Turns logging on or off. If on, information is sent to the file /cacheinstall/bin/CD###.log for Studio debugging purposes. This file can become very large. Use carefully. For details, see "Logging" in *Using Caché Direct*.
Ctrl+Shift+A	Debug Attach Attach the debugger to a process.
F9	Debug Toggle Breakpoint on Current Line
Ctrl+F5	Debug Start
Ctrl+Shift+F5	Debug Restart
Ctrl+F10	Debug Run to Cursor Resumes program execution and pauses at the cursor line or a breakpoint.
F11	Debug Step Into From a break or an interrupt, step into the next loop.
Shift+F11	Debug Step Out Step out of the current process.
F10	Debug Step Over Skip over the next process.
Shift+F5	Debug Stop
Templates	
Ctrl+Shift+1 - Ctrl+Shift+9	Open Template Can be used as accelerators for Studio Templates. To set an accelerator, add an attribute in the form `accelerator="#"` to the template (in either the csp:StudioInteractiveTemplate tag or the csp:StudioSimpleTemplate tag). This sets an accelerator of **Ctrl+Shift+#** for the template. The number (#) can be 0-9.
Ctrl+T	Open Templates
Wizards: Arguments for New Method wizard and Parameters for New Query wizard	
Alt+A	Add

Accelerator	Action
Alt+R	Remove
Alt+U	Up
Alt+D	Down

19.13.1 Inserting MultiValue Characters

You can insert the following four MultiValue characters by pressing **Ctrl+M** and then pressing one of the characters shown in the table below.

Ctrl+M [	Insert MultiValue textmark. Press Ctrl+M, then [.
Ctrl+M]	Insert MultiValue subvaluemark. Press Ctrl+M, then].
Ctrl+M \	Insert MultiValue valuemark. Press Ctrl+M, then \\.
Ctrl+M ^	Insert MultiValue attributemark. Press Ctrl+M, then ^.

*The German and Swiss keyboards have some different arrangements than most other keyboards. This limits some of the available keyboard accelerator combinations due to the AltGr key and the way key presses are reported in Windows.

19.14 Adding to a Studio Menu

To add a menu item to a Studio menu,

1. In the Studio toolbar, right-click the menu name and select **Customize**.
2. Select the Tools tab, add the.exe file.

20

Setting Studio Options

You can modify the behavior of aspects of Studio by selecting **Tools > Options**.

The options dialog contains tabs described in the following sections.

20.1 Environment Options

These options control the Studio environment:

General

Preferred Language: Sets the default language version used to create new classes.

Items shown in recently used lists: Specifies the number of items displayed in the most recently used file list (under the **File** menu).

Open File Added to Project: If enabled, adding a file to the current project automatically opens the file in Studio.

Tabbed Document Selector: If enabled, displays a tab for each open document. You can choose whether the row displays on the top of bottom of the Studio window.

When connected show documents from last time: If selected, when you start Studio all documents that were open the last time you were in the current namespace are reopened. To bypass this option, hold down **Shift** when Studio opens.

Hide Find and Replace window after operation complete: If selected, the Find and Replace dialog exits when it completes it task.

Compare: Select a document to compare this document to using **Tools > Compare**.

Font

Specifies the typeface and size of the fonts used in the following windows and print. Each can use any Windows font but it is limited to a single typeface and size: **Editor Window**, **Output Window**, **Print**, **Workspace**, **Inspector**.

Keyboard

Show commands containing: Specify a command to search for.

Shortcut for selected command: Shows Whether the selected command has a shortcut key assigned to it.

Press new shortcut key: Enter a shortcut key to assign to the selected command.

Short currently used by: Shows whether the shortcut key you entered is currently assigned to another command.

Reset All Resets all keyboard shortcuts to original settings.

Remove Removes an existing shortcut key for the selected command.

Open Dialog

Automatically apply last mask. If checked, the last search mask is used automatically in the **File > Open** dialog. Selected by default.

Use additional dedicated server process. Recommended only on very large systems if user want option to abort data collection. In rare situations on very large systems, searching for a file in the **File > Open** dialog can take a significant amount of time. To solve this, check this option to run the search in a separate process. If the search take more than .2 seconds, Studio displays a progress bar with a **Cancel** button. Note that if you check this box and an additional process is started, it affects the license count.

Server defined colors

You can select a color for the status panel background or the document background for a software instance. Display a color palette by selecting the square to the far right of the instance. Select a color from the palette, which is shown by a color swatch and its hexadecimal color code.

Documentation and Proxy

HTTP Address used to serve on-line documentation: Specifies the location Studio uses to fetch on-line documentation. Select **Automatic** to use the default location associated with the instance, or **HTTP Address** to supply a different location.

Templates and add-ins will use Proxy server for 'Caché_instance': Specify the server address and port number of a Proxy server to load templates and add-ins for the current instance. The **Address** field supports the address formats `http://` and `https://`, as well as address such as `localhost`. If no `://` detected in the address, Studio adds `http://`.

Class

Multiline Method Argument: If selected, method arguments are displayed in the class editor one per line. Note that if Multiline Method Argument is enabled and you use Find in Files and then select a line in the Find in Files output of a file that has multiline method arguments, the cursor may go to the wrong line number. Disable Multiline Method Arguments if this is a problem.

Option explicit: If selected, you see a syntax error if you refer to a variable that hasn't been declared. (Select this and Studio Assist to display undeclared local variables when entering a #DIM statement in a method or a DO statement in a routine.)

Show internal class members in Studio Assist: If selected, Studio Assist lists class members marked as internal.

Track variables: If selected, a green wavy underline indicates any questionable use of a variable—specifically: A variable is used that has no value, was never created, or has already been killed. Or a variable is given a value, but never used or killed before being used.

Open class in contracted view: If selected, by default a class opens with all collapsible sections contracted (as though Ctrl+- had been pressed). If not selected, a class opens with collapsible sections open (as though Ctrl++ had been pressed).

Code Snippets

Display Snippets Check the types of code snippets you want Studio to display. You can define your own sets of code snippets by specifying a name and a text file. Use **Create Code Snippet** from the document's context (right-click) menu. The snippet is created in your currently-active user-defined set or if no set is currently-active, then in the first set.

Advanced

Auto Save prevents you from losing changes to Studio documents on a software or system failure. By default, Auto Save is enabled and saves every 5 minutes. It saves any document that is open and has been modified since the last save into a file that is a text representation of the document, C:\Documents and Settings\<username>\Local Settings\Temp\CST*.tmp. If you subsequently save (or close) the document, this .tmp file is deleted. If Studio crashes, the next time that Studio is opened, you get a message telling you that the temporary file exists. If you open this temporary document, you can paste the relevant portions into a Studio document.

Enable service status check (Recommended) determines how often Studio checks to see if open documents and/or the open project changed on the Server outside of this Studio process. If Studio is the active application, it uses the first setting, **Studio is active application (2–60 sec)**. If Studio is running in the background, it uses the second setting: **Studio is background application (30–600 sec)**. If you are on a slow sytem, you can uncheck this option. As result studio will not check server status and will not be able timely detect if documents or project on server were modified or studio lost connection. Use with caution.

Automatically reload document when it is modified on the server and it has not been edited in Studio: If selected and you have a document open in an editor, but have not yet edited it, and someone else saves a new version to the server, the file in your window is automatically updated. This setting can be enabled on a namespace basis using a global: `Set ^%SYS("Studio","Reload")=1` (or 0 to turn off).

Show generated documents in Namespace tree: If selected, the namespace window in workspace shows generated files. If not selected, generated files are not displayed.

Use INT as default for ObjectScript: If selected, when you select **File > New > Caché Objectscript Routine**, an INT file is opened. If not selected, a MAC file is opened.

Pass credentials to View Web Page: If selected, Studio checks your permissions when you select `View Web Page`.

Use default language (will cause reset toolbars and restart). If checked, Studio loads language specific resources. To override your system’s default language (that is, to see all menus in English) uncheck this box. When you accept the changes, toolbars are reset and Studio is restarted

Export flags:Enter flags that you want to use when exporting files. See theFlags and Qualifiers section of the $SYSTEM entry in the *Caché ObjectScript Reference* for more information.

20.2 Editor Options

The editor options allow you to control the behavior of the Studio text editor. These options include:

Syntax Check and Assist

Enable Syntax Checking: If enabled, syntax errors are highlighted. You can specify when syntax checking should be performed - either on each change (each character that you type or erase) (**Syntax Check on Change**) or when the cursor leaves the current line (**Syntax Check on Leave Line**).You can specify whether you also want the errors to be underlined with a wavy red line (**Underline Errors**).

Parsing delay. Check if parsing slow. Uncheck if line flashing. If **Syntax Check on Change** and **Parsing delay** are both enabled and you are entering text faster than the parser can reparse, the text flashes between black and the parsed color. If the text is flashing, disable **Parsing Delay** by clearing this option or pressing **Ctrl+Alt+O**. Response may slow slightly since every keystroke causes a reparse, but the flashing stops. This switch needs to be set at the start of each Studio session.

Enable Bracket Matching: If enabled, pairs of matching brackets enclosing the current cursor point are bolded. Depending on the language you are in, *brackets* include [ ] square brackets, ( ) parentheses, and < > angle brackets. Note that for **Enable Bracket Matching** to work, **Enable Syntax Checking** must be checked. **Bracket Matching Line Limit**Limits the number of lines to search above and below the caret position to locate a matching bracket (as an unlimited search in a long file would slow the editor down significantly).

Studio Assist: Enables *code completion*. As you are entering ObjectScript code, a drop-down menu is displayed showing available options for what you can enter next. If you type a package name, available classes are listed. If you type a class, available methods are listed. If you type a method, available arguments are listed. Available options may be listed in other locations as well, such as with **#dim** declarations and trigger code.

To display undeclared local variables when entering a #DIM statement, you must be in a method, and you must have selected Studio Assist and Option Explicit. To be listed, a variable must not begin with %, must not be a parameter or in the public list, and must not have been declared already.))

If you type $$$, available macros are listed as follows: User-defined macros are listed if they are defined in the current file and if they are defined in an include file and, within the include file, they are preceded by a line beginning with three slashes, ///. System-defined macros are listed if the current file is a class file.

Following a partial member name, Studio displays a list of matching members as follows: If the partial entry begins with double-quotes (or a single quote), the popup contain only members whose names must be quoted in the program; that is, they contain spaces or other non-alphanumeric characters. If the partial entry does NOT begin with double-quotes, the popup contains only members whose names do not need to be quoted. If Studio Assist is triggered directly after a period, the popup contains all member names.

Colors

The Studio syntax checker uses a different coloring scheme for each language it supports. This option lets you specify the colors used to highlight syntactic elements when Studio syntax coloring is enabled.

To change the color used by the Studio Editor for a specific syntax element do the following in the Options dialog Appearance tab:

1. Select a language (such as ObjectScript) from the available options
2. Select a syntax element (such as comments)
3. Select the desired foreground (and background color if you must!) color
4. Select **Apply** to use the new color scheme.

Reset reverts the selected syntax element to its default color.

Reset All reverts all syntax colors to their default values.

Note: You can also change the color for a particular syntax element by right-clicking it in the editor window and selecting **Set Syntax Color.**

Keyword Expansion Case

This feature only applies to ObjectScript routines.

Specifies the case (**Use Current Case**, **Uppercase**, **Lowercase**, or **Mixed Case**) used to expand ObjectScript commands when you select **Edit > Advanced > Expand Commands**. Set this option, highlight the code you want to expand, then select **Edit > Advanced > Expand Commands**. This also applies if you are compressing commands.

Indentation

Defines characteristics of automatic indentation.

- **Basic**: If enabled, if a line begins with a tab, a space, or any combination of spaces and tabs, when you press Enter, the next line is started automatically with the same combination of spaces and tabs.
- **User-defined ('/t' for tab)**

  : If enabled, you can specify any characters that you would like to be automatically entered at the start of each subsequent line. For example, if you enter the set of characters `\t.#/;` (tab, dot, pound, slash, semi-colon) and if a line begins with any of these characters or any combination of these characters, when you press Enter, the next line is started automatically with the same combination of characters.
- **None**: No automatic indentation is provided.

Comment

Displays a table of comment delimiters for Studio document types. Select in a cell to enter a delimiter. Highlight a block of text in a Studio document and press **Ctrl + Alt + /** to delimit the block with Multi-Start and Multi-End characters.

View

Controls the display of some items.

- **Show Special Characters**: If enabled, the editor displays newline and tab characters using special symbols.
- **Show Line Numbers**: If enabled, the editor displays line numbers.
- **Tab Size**: Specifies the size of a tab by number of spaces.
- **Convert tabs to spaces**: If enabled, the editor converts tabs to spaces.
- **Cloudy background color for generated files**: If enabled, the editor displays generated files with a greyed background color to differentiate them from user-created files.

20.3 Compiler Options

These options affect how Studio compiles your code. There are two pages, **Flags and Optimization** and **Behavior**.

Flags and Optimization

This page includes 3 sections: **Compile Flags**, **Optimization Level**, and **Flags**.

The **Compile Flags** section includes

Keep Generated Source Code: If enabled, specifies that the compiler should not delete any intermediate source code (routines) that it creates as a consequence of compiling.

Compile Dependent Classes: If enabled, the compiler compiles all of a class' dependent subclasses.

Skip Related Up-to-date Documents: If enabled, this sets the “Do not compile up-to-date documents” flag and the compiler does not compile related documents that have not been modified since their last compilation. The document that is current in the editor is always, however, recompiled.

Compile In-use Classes: If enabled, the compiler compiles a class even if there currently are instances of the class in active use.

In the **Optimization Level** section, you can set the level of optimization to improve execution speed. If optimization is enabled, the compiler reorganizes the code for maximum benefit, including the copying of expressions between classes to eliminate method calls. Levels are:

- **No optimization**: Recommended during development. It does not recompile dependent classes and it keeps a strong correspondence between source and object code so it is easier to read and debug. Default.
- **Optimize properties**: Optimizes any reference to ..property to the instance variable reference (for simple properties described by datatypes where the **get**/**set** method is not overridden).
- **Optimize within class and calls to library classes**: Optimizes classes, as well as calls to system (%) classes (as code may be extracted and moved during the process). Note that incremental compile no longer works for optimized classes.

In the **Flags** field, enter compiler flags you want used in this field shown in the table.

To see this list of flags in the terminal, enter: `d ##class(%SYSTEM.OBJ).ShowFlags()`

To see a list of qualifiers, enter: `d ##class(%SYSTEM.OBJ).ShowQualifiers()`

Flag	Effect
a	Include application classes. This flag is set by default.
b	Include sub classes.
c	Compile. Compile the class definition(s) after loading.
d	Display. This flag is set by default.
e	Delete extent.
h	Generate help.
I	Validate XML export format against schema on Load.
k	Keep source. When this flag is set, source code of generated routines is kept.
l	Lock classes while compiling. This flag is set by default.
p	Percent. Include classes with names of the form %*.
r	Recursive. Compile all the classes that are dependency predecessors.
s	Process system messages or application messages.
u	Update only. Skip compilation of classes that are already up-to-date.
v	Include classes that are related to the current class in the way that they either reference to or are referenced by the current class in SQL usage.

Behavior

- **Before Compile**: You can choose a default behavior for Studio to take when you select **Compile**. You can select that, before compiling, Studio will **automatically save all modified documents**, **prompt to save modified documents**, or **do not save modified documents**.

- **Compile Routine on Save** Select this option to have the system compile any modified documents when you select **Save**. By default, this option is turned off.

20.4 SQL Options

Use these options if you primarily use Studio to create classes that map onto existing legacy data.

Legacy Mode: Enable Legacy SQL Mode For Classes

If enabled, the other default settings on this tab are enabled. This option effects only how Studio wizards operate; it has nothing to do with the runtime behavior of applications.

Default Storage Type: Specifies the storage class used when the New Class wizard creates a new class.

Default $Piece Separator: Specifies the default data delimiter used when defining a mapping to legacy data structures.

Default Collation: Specifies the default index collation used when defining a mapping to legacy data structures.

Private Row ID: Specifies whether new classes should have their SqlRowIdPrivate flag set by default.

Automatically Generate Row ID: Automatically creates a row id field when mapping data to existing storage structures.

20.5 Studio Look Options

Select a theme from the list to change the color scheme of Studio using this option.

A

Using Studio Source Control Hooks

To place Caché code under source control, you need to connect Studio to a third-party source control system. This appendix describes how to do this. It discusses the following topics:

- The Overview provides a summary of how to place Caché code under source control
- An introduction to Caché documents (class definitions, routines, include files, and CSP files), tools that Caché provides to manage documents and files, and some issues to consider when mapping Caché documents to XML files
- How to create and activate a source control class, in general
- How to execute the functions or methods of your source control software
- A look at the source control sample provided in the SAMPLES namespace

A.1 Overview

To place a Caché development project under source control, do the following:

- Represent units of code as XML files and write them to a document system. Caché considers a class definition, a routine, or a CSP file as a single unit called a *document*.
- Place the XML files under source control.
- Ensure that the XML files are kept synchronized with the Caché documents (and vice versa), and make sure that both are kept in the appropriate read-write state.
- Ensure that you can perform source control activities from within Studio.
- Ensure that Studio always has the same information that the source control system has as to the status of a document: whether the document has been checked out, and, if checked out, by whom.

A.2 Caché Documents

A Caché document is a class definition, a routine, an include file, or a CSP file. Caché records information about each Caché document, such as whether it has changed since the last compilation. Your source control system treats each Caché document as a separate unit. The state of a document is shown by an icon in the document window.

In Caché, you work within one namespace at a time. The same is true for your source control system.

A.2.1 Tools for Managing Documents and Files

Caché provides the following tools for managing Caché documents and external files:

- The %Studio.Extension.Base and %Studio.SourceControl.Base classes provide methods for basic document management. You can extend one of these classes to add menu items that act on Caché documents. These classes are discussed in the section "Creating and Activating a Source Control Class" in this book.
- The **$system.OBJ.Export** function exports a Caché document to an XML file in the external document system. This XML file contains all the information needed to reconstruct the Caché document. For example, for a class document, the corresponding XML file is a text representation of the entire class definition, which includes all code, properties, comments, and so on.
- The **$system.OBJ.Load** function loads an external XML file and overwrites the corresponding Caché document, if one exists.
- The **%RoutineMgr.TS** class method returns the timestamp for a Caché document. This method also returns, by reference, the compile time for the Caché document, as the second argument.

A.2.2 Deciding How to Map Internal and External Names

Each document has two names:

- An internal name, the name you use in the **Open** dialog box in Studio.
- An external name, which should be the complete external file name, including path. Because of differences between supported Caché platforms, it is not possible to provide a meaningful default.

You will set up a bidirectional mapping between the internal names and the external names. In practice, deciding how to do this may be one of the most challenging parts of creating a source control interface. This mapping is customer-specific and should be considered carefully.

You want the source control tool to group similar items. For example, the sample uses the following directory structure:

- Class files are in the cls subdirectory, which contains subdirectories corresponding to the package hierarchy of the classes.
- .INT routines are in the int subdirectory.
- .MAC routines are in the mac subdirectory.
- CSP files are in the csp subdirectory, which contains subdirectories corresponding to the package hierarchy of the CSP files.

For example, the external name for the class MyApp.Addresses.HomeAddress is C:\sources\cls\MyApp\Addresses\HomeAddress.xml.

This approach might be problematic if you had large numbers of routines. In such a case, you might prefer to group routines into subdirectories in some manner, perhaps by function.

A.3 Creating and Activating a Source Control Class

This section describes the basic requirements for creating and activating a source control class.

A.3.1 Extending Studio

Caché provides classes that you can use to add menu items to Studio. To add a source control menu to Studio, you would use either %Studio.Extension.Base or %Studio.SourceControl.Base.

Note: Limit on how many menus you can add to Studio: You can add up two menus with 19 menu items each.

The %Studio.Extension.Base class provides the following methods, which all use the internal name of the Caché document:

- Empty **Login** and **Logout** methods that you can implement as needed. The variable *$username* records the current user. (In the **Login** method, the *Username* argument is provided for backward compatibility; it is recommended that you use the variable *$username* instead.)
- Basic methods to indicate the status of a given Caché document: **GetStatus**, and **IsInSourceControl**. Implement these methods as needed.
- Callback methods that are executed when a user in Studio performs some action on a Caché document. These methods include **OnBeforeLoad**, **OnAfterLoad**, **OnBeforeCompile**, **OnAfterCompile**, **ItemIconState**, and so on.

Note: Studio compiles processes in separate threads. If you set properties in %Studio.Extension.Base, they may not be accessible in subsequent calls, as they may be running in different object instances. Do not use a properties to pass information from **MenuItem** to **OnBeforeCompile**. Instead, use a temporary global.

The %Studio.SourceControl.Base class is a subclass of the preceding class. %Studio.SourceControl.Base provides the following additional elements:

- An XDATA block named `Menu` that defines an additional menu for Studio: **Source Control**. By default, this menu contains the menu items **Check In**, **Check Out**, **Undo Check Out**, **Get Latest**, and **Add To Source Control**. This XDATA block also defines additional menu items for the context menu in Studio.

  All these menu items call methods also defined in this class.
- Methods named **CheckIn**, **CheckOut**, **UndoCheckOut**, **GetLatest**, and **AddToSourceControl**, which do nothing by default.

To extend Studio, you define a new class that extends one of these classes. As you see in "Activating a Source Control Class," the Management Portal provides a way to indicate which extension class is currently active in a given namespace. If an extension class is active in a given namespace, and if that class defines an XDATA menu block, those menu items are added to Studio.

A.3.2 Creating a Source Control Class

To create a source control class, do the following:

1. Create a subclass of %Studio.Extension.Base or %Studio.SourceControl.Base.
2. If you started with %Studio.Extension.Base, create an XDATA block named `Menu` in your subclass. (Copy and paste from %Studio.SourceControl.Base to start this.)
3. Implement the methods of this class as needed: **AddToSourceControl**, **CheckIn**, **CheckOut**, and so on. These methods would typically do the following, at a minimum:
   - If appropriate, import or export the Caché document to XML.
   - Call the appropriate function or method of your source control software, to act on the XML file.
   - Update internal information in Caché about the status of the given file.
   - Control whether the Caché document is editable.

The details depend upon the source control system. The sample demonstrates some useful techniques. See the section "Sample Source Control Class" in this book.

4. Implement the **GetStatus** method of your source control class. This is required. You might also need to implement the **IsInSourceControl** method, if the default implementation is not suitable.

A.3.3 Activating a Source Control Class

To activate a source control class for a given namespace, do the following:

1. Use the Management Portal to specify which extension class, if any, Studio should use for a given namespace. To specify the class to use:
    a. Navigate to **System Administration** > **Configuration** > **Additional Settings** > **Source Control** on the Management Portal.
    b. On the left, select the namespace to which this setting should apply.
    c. Select the name of the extension class to use (or select **NONE**) and select OK.

       This list includes all compiled subclasses of %Studio.Extension.Base.
2. If Studio is currently open, close it and reopen it, or switch to another namespace and then switch back.

A.4 Accessing Your Source Control System

The API for your source control system provides methods or functions to perform source control activities such as checking files out. Your source control class will need to make the appropriate calls to this API, and the Caché server will need to be able to locate the shared library or other file that defines the API itself.

If the source control system provides a COM interface, you can generate a set of Caché wrapper classes that you can use to call methods in that interface. To do so, you use the Caché Activate Wizard in Studio. Given an interface and the name of the package to contain the classes, the wizard generates the classes. For information, see *Using the Caché ActiveX Gateway*.

Also, it is important to remember that Caché will execute the source control commands on the Caché server. This means that your XML files will be on the Caché server, and your file mapping must work on the operating system used on that server.

A.4.1 Example 1

For the following fragment, we have used the Caché Activate Wizard to generate wrapper methods for the API for VSS. Then we can include code like the following within your source control methods:

```
do ..VSSFile.CheckIn(..VSSFile.LocalSpec,Description)
```

The details depend on the source control software, its API, and your needs.

A.4.2 Example 2

The following fragment uses a Windows command-line interface to check out a file. In this example, the source control system is Perforce:

```
/// Check this routine/class/csp file out of source control.
Method CheckOut(IntName As %String, Description As %String) As %Status
{
  Set file=..ExternalName(IntName)
  If file="" Quit $$$OK
  //...
 Set cmd="p4 edit """_file_""""


  #; execute the actual command
  Set sc=..RunCmd(cmd)
  If $$$ISERR(sc) Quit sc

  #; If the file still does not exist or
  #; if it is not writable then checkout failed
  If '##class(%File).Exists(file)||(##class(%File).ReadOnly(file)) {
    Quit $$$ERROR($$$GeneralError,
                  "Failure: '"_IntName_"' not writeable in file sys")
  }

  #; make sure we have latest version
  Set sc=..OnBeforeLoad(IntName)
  If $$$ISERR(sc) Quit sc

  //...
  Quit sc
}
```

In this example, **RunCmd** is another method, which executes the given command and does some generic error checking. (**RunCmd** issues the OS command via the **$ZF(-1)** interface.)

Also, this **CheckOut** method calls the **OnBeforeLoad** method, which ensures that the Caché document and the external XML file are synchronized.

A.5 Sample Source Control Class

The SAMPLES namespace provides a sample source control class, Studio.SourceControl.Example. This section shows how this sample works. The following topics are discussed:

- An introduction to the sample and its assumptions
- The global in which Studio.SourceControl.Example records needed information
- How the class determines the external names of the Caché documents, which are used as the XML file names
- How the class synchronizes the Caché document and the corresponding XML file
- How the class implements the required GetStatus method, which ensures that the Caché document is read-only when appropriate
- Details of the actual source control methods in the class
- Other notes about this class

Note: The class in your SAMPLES namespace could be slightly different from the examples shown here. In particular, some of the line breaks have been adjusted for readability in this document.

A.5.1 Introduction

Studio.SourceControl.Example is a partial example that does not make any calls to a source control system. It simply maintains external XML files that such a system would use. Despite this simplification, however, the sample demonstrates all the following:

- Establishing and maintaining a relationship between each Caché document and a corresponding external file.

- Keeping Caché up-to-date with the status of each file.
- Appropriately controlling the read-write state of each Caché document.
- Defining methods to check files in and out.

Note that the sample does not modify the read-write state of the external files; the source control system would be responsible for that. Also, the sample implements only the **Check In** and **Check Out** methods.

To try this example in the SAMPLES namespace, do the following:

1. Use the Management Portal to enable this source control class (Studio.SourceControl.Example), as described earlier in "Activating a Source Control Class."
2. In the Studio **Workspace** window, double-click a Caché document. Notice a message like the following in the **Output** window:

   ```
   File C:\sources\cls\User\LotteryUser.xml not found, skipping import
   ```

3. Edit the document (for example by adding a comment).
4. Select **File —> Save**. You will see a message like the following in the **Output** window:

   ```
   Exported 'User.LotteryActivity.CLS' to file
   'C:\sources\cls\User\LotteryActivity.xml'
   ```

   At this step, you have implicitly added the Caché document to the source control system.
5. Try to make another edit. The Studio displays a dialog box that asks if you want to check the file out. Select **No**. Notice that the Caché document remains read-only.
6. Select **Source Control —> Check Out** and then select **Yes**. You can now edit the Caché document.
7. Select **Source Control —> Check In** and then select **Yes**. The Caché document is now read-only again.

Other menu items on the **Source Control** menu do nothing, because the sample implements only the **Check In** and **Check Out** methods.

A.5.2 Global

The Studio.SourceControl.Example sample uses a global to record any needed persistent information. Methods in this class maintain and use the `^MySourceControl` global, which has the following structure:

Node	Contents
`^MySourceControl("base")`	The absolute path of the directory that will store the XML files. The default is C:\sources\
`^MySourceControl(0,`*IntName*`)`, where *IntName* is the internal name of a Caché file	The date and time when the corresponding external file was last modified
`^MySourceControl(1,`*IntName*`)`	The date and time when this Caché document was last modified
`^MySourceControl(2,`*IntName*`)`	The name of the user who has this Caché document checked out, if any

This global is purely a sample and is used only by this class.

A.5.3 Determining the External Names

If you enable the Studio.SourceControl.Example class, it maintains external XML files that correspond to any Caché document that you load or create. It writes these files to the directory C:\sources\ by default, as described in "Deciding How to Map Internal and External Names." For example, the external name for the class MyApp.Addresses.HomeAddress is C:\sources\cls\MyApp\Addresses\HomeAddress.xml.

Within the sample, the **ExternalName** method determines the external file name for any Caché document. This method is as follows:

```
Method ExternalName(IntName As %String) As %String
{
 Set name=$piece(IntName,".",1,$length(IntName,".")-1)
 Set ext=$zconvert($piece(IntName,".",$length(IntName,".")),"l")
 If name="" Quit ""
 Set filename=ext_"\"_$translate(name,".","\")_".xml"
 Quit $get(^MySourceControl("base"),"C:\sources\")_filename
}
```

The sample is suitable only for Windows, of course.

A.5.4 Synchronizing the Caché Document and the External File

Two methods are responsible for ensuring that the Caché document and the corresponding XML file are kept synchronized with each other:

- Studio calls the **OnBeforeLoad** method immediately before loading any Caché document into the work space.
- Studio calls the **OnAfterSave** method immediately after saving any Caché document.

In the sample, the **OnBeforeLoad** method is as follows:

```
Method OnBeforeLoad(IntName As %String) As %Status
{
 Set filename=..ExternalName(IntName)
 If filename="" Quit $$$OK

 #; If no file then skip the import
 If '##class(%File).Exists(filename) {
     Write !,"File ",filename," not found, skipping import"
     Quit $$$OK
 }

 #; If the timestamp on the file is the same as the last time
 #; it was imported, then do nothing
 If ##class(%File).GetFileDateModified(filename)=
                  $get(^MySourceControl(0,IntName)) {
     Quit $$$OK
 }

 #; Call the function to do the load
 Set sc=$system.OBJ.Load(filename,"-l-d")
 If $$$ISOK(sc) {
 Write !,"Imported '",IntName,"' from file '",filename,"'"
 Set ^MySourceControl(0,IntName)=##class(%File).GetFileDateModified(filename)
 Set ^MySourceControl(1,IntName)=##class(%RoutineMgr).TS(IntName)
 } Else {
 Do $SYSTEM.Status.DecomposeStatus(sc,.errors,"d")
 }
 Quit sc
}
```

Note:

- The method checks to see whether the external file exists yet. If the external file exists, the method compares its time stamp to the time stamp of the Caché document. If the external file is more recent than the Caché document, the method loads it.

It is not strictly necessary to check the time stamps; this method could load the external document every time. The check is performed because it offers a performance improvement.

- The method sets the relevant nodes of the `^MySourceControl` global.

The **OnAfterSave** method is analogous, as you can see in the sample itself.

Note: Not only does Studio call these methods automatically as noted above, we will call these methods whenever we need to ensure that the Caché document and the external document are synchronized.

A.5.5 Controlling the Status of the Caché Document

The **GetStatus** method of your source control class is responsible for returning information about the status of the given Caché document. This method has the following signature:

```
Method GetStatus(IntName As %String,
                 ByRef IsInSourceControl As %Boolean,
                 ByRef Editable As %Boolean,
                 ByRef IsCheckedOut As %Boolean,
                 ByRef UserCheckedOut As %String) As %Status
```

Studio calls this method at various times when you work with a Caché document. It uses this method to determine if a Caché document is read-only, for example. When you implement a source control class, you must implement this method appropriately.

In the sample, this method is implemented as follows:

```
Method GetStatus(IntName As %String,
                 ByRef IsInSourceControl As %Boolean,
                 ByRef Editable As %Boolean,
                 ByRef IsCheckedOut As %Boolean,
                 ByRef UserCheckedOut As %String) As %Status
{
 Set Editable=0,IsCheckedOut=0,UserCheckedOut=""
 Set filename=..ExternalName(IntName)
 Set IsInSourceControl=(filename'=""&&(##class(%document).Exists(filename)))
 If 'IsInSourceControl Set Editable=1 Quit $$$OK

 If $data(^MySourceControl(2,IntName))
 {Set IsCheckedOut=1
 Set UserCheckedOut=$listget(^MySourceControl(2,IntName))}

 If IsCheckedOut,UserCheckedOut=..Username Set Editable=1
 Quit ..OnBeforeLoad(IntName)
}
```

Here is how this method works:

1. It first initializes all the arguments that it returns by reference.
2. The method then checks to see whether the external document exists yet; if it does not, the Caché document should be editable.
3. The method then checks the `^MySourceControl` global to see if anyone has checked this document out. If so, and if that user is the current user, the document is editable. If the document is checked out to a different user, it is uneditable to the current user.
4. Finally, the method calls the **OnBeforeLoad** method, which was described earlier in this document. This step ensures that the Caché document and the external XML file are synchronized and that the relevant nodes of the `^MySourceControl` global get set.

A.5.6 Source Control Actions

The sample implements methods for the two most basic source actions: check in and check out.

The **CheckIn** method is as follows:

```
Method CheckIn(IntName As %String, Description As %String) As %Status
{
 #; See if we have it checked out
 If '$data(^MySourceControl(2,IntName)) {
   Quit $$$ERROR($$$GeneralError,"You cannot check in an item
                                  you have not checked out")
 }
 If $listget(^MySourceControl(2,IntName))'=..Username {
   Quit $$$ERROR($$$GeneralError,"User '"_
         $listget(^MySourceControl(2,IntName))_"'has this item checked out")
}

 #; Write out the latest version
 Set sc=..OnAfterSave(IntName)
 If $$$ISERR(sc) Quit sc

 #; Remove the global to show that we have checked it in
 Kill ^MySourceControl(2,IntName)
 Quit $$$OK
}
```

Notes:

- This method uses the `^MySourceControl` global to see whether the current user can actually check this Caché document in.
- If the user can check the document out, the method does the following:
  - The method calls **OnAfterSave** to make sure that the Caché document and the external document are synchronized.
  - The method kills the appropriate node of the `^MySourceControl` global to indicate that the document is now checked in.

The **CheckOut** method is analogous.

These methods could be extended to include the appropriate calls to a third-party source control system.

A.5.7 Other Details

By default, the method **IsInSourceControl** calls the **GetStatus** method and gets the needed information from there.

In the sample, the method **IsInSourceControl** returns true for all internal names; recall that all documents are assumed to be under source control.

A class definition can be changed when you compile it, because compilation can update the storage information. Accordingly, the sample implements the **OnAfterCompile** method. This method just calls the **OnAfterSave** method, because it needs the same logic as that method provides; specifically, it needs to check whether the Caché document has changed and if so, save the XML file again.

We do not recommend using process private globals in source control hooks because processes may not run in the same thread. For more information, see the chapter “Cache 2012.1” in the *Caché Release Notes and Upgrade Checklist Archive*.

B

Frequently Asked Questions About Studio

A Question and Answer Set about Studio.

Projects

What is a project?

A project is a collection of class definitions, routines, and/or CSP files that you can group together for the sake of convenience.

Using projects gives you an easy way to return to your work when you start a Studio session. For example, you can place all the classes related to an application, or part of an application, in a project. When you start Studio, open this project and the Project tab of the Workspace window displays all the classes in a convenient list.

You can also export and import entire projects to and from a single external file making it easy to save or pass around application code.

How do I add an item to a project?

Here are some of the ways to add items to a the current project:

- Before opening one or more items (with **File > Open**), select the **Add to Project** check box in the **Open** dialog.
- To add the item in the current editor window to the current project, select **Project > Add Item**.
- In the workspace window, highlight an item, right-click, and select **Add to Project**.

Can I add something from another namespace to my project?

No. A project can only contain items that are visible to the current Caché namespace.

Can an item belong to multiple projects?

Yes. A project is a specified set of items (class definitions, routines, and CSP files) that you choose to group together. The items themselves have no link back to the projects they may belong to. There is no limit to how many projects an item can belong to.

What if I don't want to use projects?

You are not required to use projects with Studio; you can completely ignore them if you like. To ignore projects, do not add any items to the default project and ignore the prompt asking you if you want to save your project when you exit Studio.

Can I export a project?

Yes. Select **Tools > Export > Export Project**. Enter a file name and press **OK**. This exports the entire contents of the current project (including the project definition) to a single XML file.

How do I delete a project?

Select **File > Open** to list all your projects. Right-click a project and select **Delete**.

Note that you can use **File > Open** to delete any type of item on the server in this way.

Opening Files

How do I open a class definition?

To open an existing class definition (that is, one saved on the Caché server), do the following:

1. Make sure you are connected to the Caché namespace and server containing the class definition.
2. Select **File > Open**.
3. In the **Open** dialog, make sure that class definitions are listed by selecting `Class Definitions` (.CLS) or `All` in the **File Types** combo box.
4. Package names are listed in the file list as folders. Select a package name to list all the classes (or subpackages) within the package. Double-click a class name to open it.
5. Alternatively, you can enter the name of the class you want directly into the filename edit box with a .cls extension (such as Sample.Person.cls) and select **Open**.

How do I open a routine?

To open an existing routine (that is, one saved in the Caché server), do the following:

1. Make sure you are connected to the Caché namespace and server containing the routine.
2. Select **File > Open**.
3. In the **Open** dialog, make sure that routines are listed by selecting either `MAC routines` (.MAC), `INT routines` (.INT), or `All` in the File Types combo box. Double-click a routine name.
4. Alternatively, you can enter a routine name with extension directly into the filename edit box (such as MyRoutine.MAC) and select **Open**.

How do I open a CSP file?

You can open a CSP file in the same way that you open a class definition or a routine. The main difference is that the **Open** dialog lists CSP Applications (for example, /csp/samples) as folders; select the name of an application to see the CSP pages within it.

What does the Show System check box in the Open dialog do?

If the **Show System** check box is selected, then the **File > Open** dialog lists system items (items whose names start with the % character and are stored in the CACHELIB database) along with items in the current namespace.

Can I use pattern matching in the File > Open dialog?

Yes. You can use the "*" character as a wildcard to match any number of any character as you can in a standard **File > Open** dialog. You can use file extensions to filter certain items; for example, "*.cls" lists all Class Definitions in the selected package. You can use the "?" character to match any character. Note that these are Windows pattern matching conventions, not Caché pattern matching.

How do I open a routine from a different namespace?

The Studio **File > Open** dialog lists items from only the current namespace and server. To open a routine from a different namespace or server:

1. Select **File > Change Namespace**.
2. Open the desired routine.

Can I open a % class?

Yes. You can list % classes (classes whose package name starts with a % character and are stored within the CACHELIB database) from the **File > Open** dialog by selecting the **Show System** check box at the bottom.

Studio opens % classes as read-only if you open them while connected to a namespace other than %SYS.

What does File > Connect do?

Studio maintains a connection to a specific Caché namespace and server. It uses this connection to provide a list of classes (such as for specifying property types, super classes, etc.). It also uses this connection for debugging. **File > Connect** lets you connect to a different server.

Debugging

How do I start the debugger?

You can connect the debugger to a target process in of the following ways:

- Define a "debugging target" (name of program or routine to debug) for the current project with **Project > Settings**. Then select **Debug > Go** to start the target program and connect to its server process.
- Select **Debug > Attach** to select from a list of running processes on a Caché server to connect to.

For more details refer to the chapter “Using the Studio Debugger” in this book.

How can I debug a class?

The Studio debugger lets you step through the execution of programs running on a Caché server. These programs can include INT files, MAC files, methods within CLS files, CSP classes responding to HTTP requests, server-side methods invoked from Java or ActiveX clients, or server-hosted applications.

1. To view the INT file during debugging and to save the INT for further review later, set the **Keep Generated Source Code** option before you compile your class, located on the **Tools > Options > Compiler > General Flags** page.
2. Set a breakpoint at the desired location in a class method (or any of the other files mentioned above) by pressing **F9** (toggles breakpoint) on the desired source line.
3. Set a debug target to specify where you want the debugger to begin code execution using **Debug > Debug Target**. Enter the name of the class and the method that you want to step through.
4. Start the debugger with **Ctrl+F5** or **Debug > Go**.

Can I watch variables?

Yes. While debugging, enter a variable name (or an expression) in the left-hand column of the Studio Watch Window. Each time the debugger pauses, the variable or expression is reevaluated.

Editing

What do the different colors in the editor mean? Can I change the colors in the editor?

The Studio uses colors to differentiate the syntax elements of a given language.

You can change the colors used for the various syntax elements as follows:

1. Select **Tools > Options > Editor > Colors**.
2. Select a language.
3. Select an element (comment, variable, etc.)—the list of available elements depends on the selected language.
4. Select Foreground and Background colors andselect **OK**.

Why is there a wavy, red line underneath my code?

The wavy, red line indicates that the underlined code (or possibly code before it) contains syntax errors.

Does Studio support Kanji and Chinese characters?

Yes. Studio has complete support for Unicode and Kanji characters.

Does Studio support Hebrew and Arabic characters?

Yes. The Studio Editor supports Hebrew and Arabic characters, as well as bidirectional editing.

Importing Files

Can I import class definitions or routines from external files?

Yes. Select **Tools > Import**.

What is the difference between Local and Remote files?

Studio is a client/server application; the Studio itself runs on a client system and talks to a server. The server can either be on the same machine or on a remote machine. The Studio uses the terms “Local” and “Remote” to refer to operating system files (such as when you are importing or exporting) that are stored on the client and server systems, respectively.

If both the client and server are on the same system then there is no difference between Local and Remote.

Printing

Can I print from Studio?

Yes. Select **File > Print** or **File > Print Preview**.

Templates

What is a Template?

Templates are a mechanism for creating user-defined Studio add-ins. A template is a program that enters a code fragment into the current document at the current cursor point. You can customize the code fragment to your needs. See the chapter “Using Studio Templates” in this book for more information.

Is there a list of available Templates?

Yes. Select **Tools > Templates > Templates**.

Can I create a new Template?

Yes. See the chapter “Using Studio Templates” in this book.

Multiuser Support

Does Studio support development by multiple users?

Yes. You can do this in several ways:

- Set up a common Caché server system and have all developers store their code on it.
- Use local Caché servers (on the developer's system) and store source code in a source control system as exported XML files.

What happens if I try to open a class (or routine) that someone else is editing?

Studio displays a dialog stating that the class (or routine) is in use by someone else and asks you if you want to open it in read-only mode.

What if someone wants to edit a super class of a class that I am working on?

Studio does not prevent another developer from modifying the super class of a class you are working on.

While Studio could take out locks on all subclasses whenever a class is opened for editing, in practice this would be annoying and unwieldy. Instead, a development team needs to work out rules and procedures for defining and modifying super classes. This is similar to how development teams in other languages (say Java) usually work with class definitions in source control systems.

Classes

How do I create a new class?

Use the **New** command in the **File** menu and ask for a new Class Definition. This invokes the New Class Wizard.

For more details, see the chapter Class Definitions in this book.

Can I see the source code generated for my class?

Yes. You can see all the source code generated by the Class Compiler with **View > View Other Code** (available when the current window contains a class definition).

Make sure that the **Keep Generated Source Code** option is set before you compile your class. This option is located on the **Tools > Options > Compiler > General Flags** page.

When I try to compile my class, the Studio says it is up to date and does not need to be compiled. Can I force a compile to happen?

Yes. Turn off the **Skip related up-to-date documents** option. This option is located on the **Tools > Options > Compiler > General Flags** page.

Routines

How do I create an INT routine?

Create a new ObjectScript routine using the **New** command in the **File** menu and then save the new routine using a name with a .INT extension. You can create an include (.INC) file in the same fashion.

SQL

How do I define an SQL View?

Studio does not include a mechanism for defining SQL views. To do this, as well as other SQL tasks, use the Management Portal.

Source Control

Does Studio integrate with external Source Control systems?

Yes. The procedure is:

1. Create a subclass of the system-supplied class %Studio.SourceControl.Base where you implement the methods to interact with your source-control system. The class that you create is called from Studio in response to particular events and then performs the actions that you have specified.
2. In the Management Portal, navigate to **System Administration** > **Configuration** > **Additional Settings** > **Source Control**, select the site-specific source-control class from the list, and select **OK**.

At this point, Studio has been configured to interact with the source-control system. When Studio attempts to open a document, prior to opening it, the **OnBeforeLoad** method of your source-control class is invoked; typically, this method checks the timestamp on a file representation of the document and, if it is newer in the file, the method calls a Caché function to import this into the current namespace. This makes sure that the user is seeing the most up-to-date version of the file.

If you modify the file and save it, then Studio calls the **OnAfterSave** method of the source-control class, which typically exports this document to the filesystem. (This keeps these files in sync with the routines, classes, etc. that are in Caché.)

When you attempt to modify a document, Studio attempts to get a lock on it, which also triggers a call to a source control method **GetStatus**. If the file is locked in source control, Studio can then ask if you want to check it out. This triggers a call to the **CheckOut** method which performs the actions required to check the item out.

%Studio.SourceControl.Base and %Studio.Extension.Base provide a set of methods that allow you to create interactions between Studio and your source-control system that are as simple or complex as you choose.

Can I create my own hooks?

Yes. You can define hooks—code that is executed whenever items are saved to or loaded from the server. For details see the appendix “Using Studio Source Control Hooks”.

Compatibility

Can I connect a Studio client to any Caché server?

A Studio client must be running either the same version of Caché or a higher version than the Caché server that it is connecting to. Example: Caché 2015.1 Studio can connect to a Caché 2015.1 (or earlier version) server. Caché 2014.1 Studio cannot connect to a Caché 2015.1 (or later) server. This applies also to maintenance releases. Example: Caché 2014.1.2 Studio can connect to a Caché 2014.1.1 (or earlier maintenance release or version) server. Caché 2014.1.0 Studio cannot connect to a Caché 2014.1.1 (or later maintenance release or version) server.

Can I run Studio on Linux?

The Studio client only runs on Windows platforms. You can use a Windows-client to talk to any server. You can also use a partition manager, such as VMWARE, to run both Windows and Linux partitions on your development system and run Studio in the Windows partition with Caché running in the Linux partition. The only trick is to configure your networking so that the Windows partition can talk to the Linux partition via TCP/IP. Studio can also run under Windows on an Intel-based Macintosh.

Studio Implementation

Why doesn't Studio use the licensed components of Microsoft Visual Studio?

There are several reasons why we built Studio from the “ground up” instead of licensing or extending Visual Studio:

- The Studio editor uses advanced parsing technology not available within the Microsoft Studio framework.
- Microsoft cannot guarantee the compatibility of future versions of Visual Studio.

Why wasn't the Studio interface developed using Java?

At this time, the only way to get acceptable performance for the Studio editor is to use direct calls to the Windows API. While there are syntax-coloring editors developed using Java they do not offer the sophisticated multi-language parsing used by Studio and they typically require very high performance computers for decent performance.