CACHÉ CORE :

SERVER – SIDE PROGRAMMING

COPYRIGHT NOTICE

Copyright © InterSystems Corporation
1997-2002
All rights reserved
NOTICE
PROPRIETARY — CONFIDENTIAL

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

InterSystems™, InterSystems Caché™, DCP™, DTM™, DSM™, and DASL™ are trademarks of InterSystems Corporation.

DSM DDP™, VAX™, VMS™, OpenVMS™, and DEC™ are trademarks of Digital Equipment Corp.

Microsoft®, MS-DOS®, Microsoft Access®, and, Excel® are registered trademarks and Windows™, Windows NT, Visual Basic™, and Visual C++™ are trademarks of Microsoft Corporation.

ORACLE® is a registered trademark of Oracle Corporation.

For questions about any InterSystems products, contact our Worldwide Response Center:

Phone: US: +1 617 621-0700 Europe: +44 (0) 1753 830-077
Fax: US: +1 617 374-9391 Europe: +44 (0) 1753 861-311

Internet: support@intersys.com **FTP Site:** ftp.intersys.com
World Wide Web: www.intersys.com
Documentation: cache.intersys.com/contactus/doc_action_form.html
European BBS: +44 (0) 1753-853-534

INDEX

Module 1: Overview

What is Caché? 10
Completely Connected 10
The Toolset 11
The Caché Cube 11
The Caché Program Group 12
Caché Object Architect 12
Caché Studio 13
Caché SQL Manager 13
Caché Explorer 14
Caché Terminal 14
Caché Control Panel 15
Caché Configuration Manager 15
Remote System Access 16
The System Viewer 16

Module 2: Getting Started

Basic Structure 18
Databases 18
Routines 18
Globals 19
Namespaces 19
Changing Namespaces 19
Creating a Namespace 20
Connections 20
ODBC Data Source 21
Bi-Directional ODBC 21
CSP Application 22
Management 22
Error Handling 22
Exercises 24

Module 3: Tour

Example Application 27
Classes and Objects 27
The MenuItem Class 27
The Order Class 28
Packages 28
Creating a Class 28
Editing a Class Definition 29
Properties 29
The Name Property 30
The Price Property 30
The Quantity Property 31
Simple Property Parameters 31
More Property Parameters 31
Parameters 32

Methods 32
Queries 33
Saving and Compiling 33
Generated Routines and Globals 33
Compile Options 34
Class Definition Language 34
Export/Import of CDL 34
Creating The User Interface 35
Form Wizards 35
Using ObjectScript 35
Examples of Using Objects 36
Syntax Conventions 36
Concurrency 36
Success or Failure 37
OREFs and IDs 37
ID 37
OIDs 37
ObjectScript: Commands 38
ObjectScript: Objects 38
ObjectScript: Pattern Examples 38
Exercises 40

Module 4: Classes

Object-Oriented Programming 43
Encapsulation 43
Object Oriented Design 43
From Words to Classes 44
Inheritance 44
Embedded Classes 44
Kinds of Classes 44
    Registered Classes 45
    Persistent Classes 45
    Persistent Object 45
    Embedded Classes 45
    Embeddable Object 46
    Storage Options 46
    Abstract Classes 46
    Datatype Classes 47
    CSP Page Class 47
    Non-Registered Classes 47
    Referencing Classes 48
        Using the Reference 48
Linking the Classes 48
The Property Type 49
Syntax for Referencing Objects 49
SQL Projection 49
%SYSTEM.OBJ Class 49
    Using %SYSTEM.OBJ 50
Object Modeling 50
RoseLink 50
Microsoft Visual Modeler 50
Nothing Etc. 51

Nothing Object Model 51
ObjectScript: Date Function 51
Exercises 54

Module 5: Properties

Hierarchy: %Library Classes 57
Datatype Classes 57
Hierarchy: Nothing Classes 57
Properties 57
Swizzling 58
Characteristics 58
Collections 58
Lists and Arrays 58
Using Lists 59
Using Arrays 59
Lists and Arrays with SQL 59
Streams 59
Stream Properties 60
Using Streams 60
Appending to a Stream 60
Relationships 60
One-To-Many 61
Parent-To-Children 61
Setting Up Relationships 62
Cardinality 62
Using Relationships 63
Viewing Properties 63
Parent->Children IDs 63
Using the Children/Many Side 63
Many-to-Many 64
Group Example 64
Transaction Example 64
Exercises 66

Module 6: Populate

Hierarchy: Collection Classes 69
Hierarchy: Stream Classes 69
Populating Classes 69
The POPSPEC Parameter 70
More About POPSPEC 70
OnPopulate().. 70
ObjectScript: Conditionals 70
Exercises 72

Module 7: Methods

Hierarchy: Population Classes 74
Methods 74
Return Values 75
Arguments 75
Reference vs. Value 76

Characteristics 76
Instance and Class Methods 77
Code Mode 77
Code Methods 77
Generator Methods 78
A Polymorphism Example 78
The Code Window 78
Relative Dot Syntax 78
Using Relative Dot Syntax 79
ObjectScript: Formatting 79
ObjectScript: For Loops 79
Exercises 81

Module 8: Business Logic 1

Hierarchy: First Methods 83
Calculated Properties 83
“Get()” Methods 83
Methods for Calculated Properties 83
SQL Computed Fields 84
Age Example 84
Callback Methods 85
Callback Method Signatures 85
Example: %OnBeforeSave() 85
ObjectScript: String Functions 86
ObjectScript: Flow Control 86
Exercises 88

Module 8.2: Business Logic 2

Hierarchy: More Methods 92
Multi-Referencing 92
From Words to Classes 93
Supplier Object Model 93
Using the Multi-Reference 93
Syntax for Multi-Referencing 94
Defining a Multi-Reference 94
Exercises 96

Module 9: Non-Registered Classes

Hierarchy: Payment Classes 99
Non-Registered Classes 99
Datatype Classe 100
Conversion 100
Datatype Methods 100
Example: %Library.Date 100
Calling Datatype Methods 101
Writing Datatype Methods 102
Datatype Population 102
Other Non-Registered Classes 102
ObjectScript: Random Numbers 103
Exercises 105

Module 10: SQL

Hierarchy: More Classes 110
Datatype class 110
Introduction to SQL 110
SELECT Queries 110
Aggregate Functions 111
Other SQL Operations 111
DDL Examples 111
Caché SQL 111
Query Wizard 112
Parameters 112
Columns 112
Conditions 113
Order By 113
Query Details 114
Row Specification 114
Views 114
Stored Procedures 115
Using SQL 115
ResultSet Class 115
Runtime Mode 115
Embedded SQL 116
Embedded SQL SELECT Queries 116
SQLCODE and %ROWCOUNT 116
Dynamic Queries 116
Using Dynamic Queries 117
Triggers 117
ObjectScript: While Loops 117
Exercises 119

Module 11: Indexes

Hierarchy: %ResultSet Class 123
Hierarchy: Promotion Class 123
Indexes 123
Index Tab 124
Customizing an Index 124
Collation 125
Caché Uses Indexes 125
Better Generated Code 125
Exercises 127

Module 12: Caché Server Pages

Caché Server Pages 129
Programming Elements 129
How does it work? 129
Benefits 130
How do you make it work? 130
Virtual Directory 130
Caché Takes Over 131

CSP Gateway 131
Caché Configuration 131
Code Generation 132
Form Wizard Source Code 132
Dreamweaver Integration 132
The CSP Form Wizard 133
The <csp:Object> Tag 133
Naming CSP Objects 133
The %request Object 134
Dynamic Data 134
A Search From the Browser 134
An SQL Query 134
A ResultSet for a Query 135
HyperEvents 135
Error Handling 135
Useful URLs 136
Topics for CSP Course 136
Exercises 138
Exercises without Dreamweaver 141

Module 13: Visual Basic

Visual Basic 144
Factory Object 144
Visual Basic Syntax 144
Syntax Table 145
Opening/Saving an Object 145
Custom Controls 145
Result Sets in Visual Basic 145
Dynamic Queries in Visual Basic 146
Form Wizard 146
Exercises 148

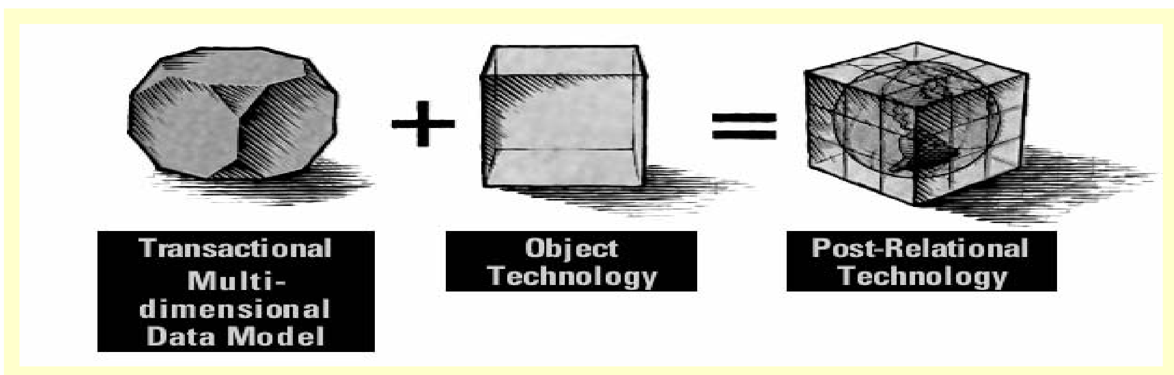
MODULE 1: OVERVIEW

1 OVERVIEW

What is Caché?

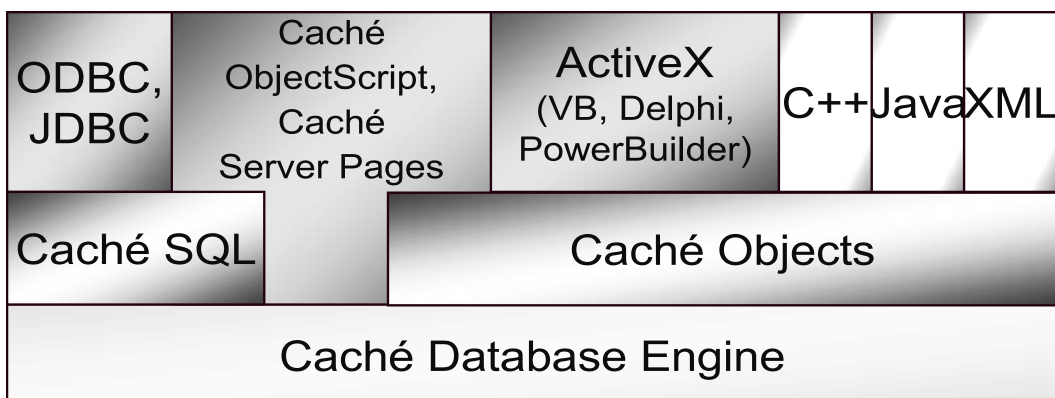
Caché is a high-performance post-relational database management system. It features:

- An object-oriented design environment;
- Integrated object and SQL access (Unified Data Architecture);
- Web development tools;
- A simple, powerful scripting language ;
- An efficient and scalable multidimensional database.



Completely Connected

Choice of tools, programming languages and data access modes.

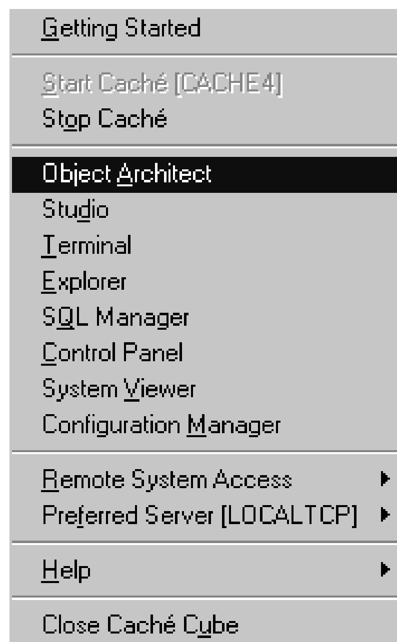


The Toolset

GUI tools: the Caché Object Architect, the Caché Studio, and the Caché SQL Manager.
Scripting language: Caché ObjectScript.
Caché Server Pages connect Web pages to the Caché database via a Web server.
Caché SQL Server allows any ODBC-compliant application or development tool to connect to Caché.
Caché SQL Gateway allows Caché to connect to any ODBC-compliant database.
Caché Object Servers expose Caché objects to ActiveX or Java.

The Caché Cube

To access the GUI tools, use the Caché cube.
The Caché cube icon appears in the system tray.
Right-click on the cube to see the Start menu.
The option to stop and start Caché is available only for the local system.
The other GUI tools are also available for remote systems.

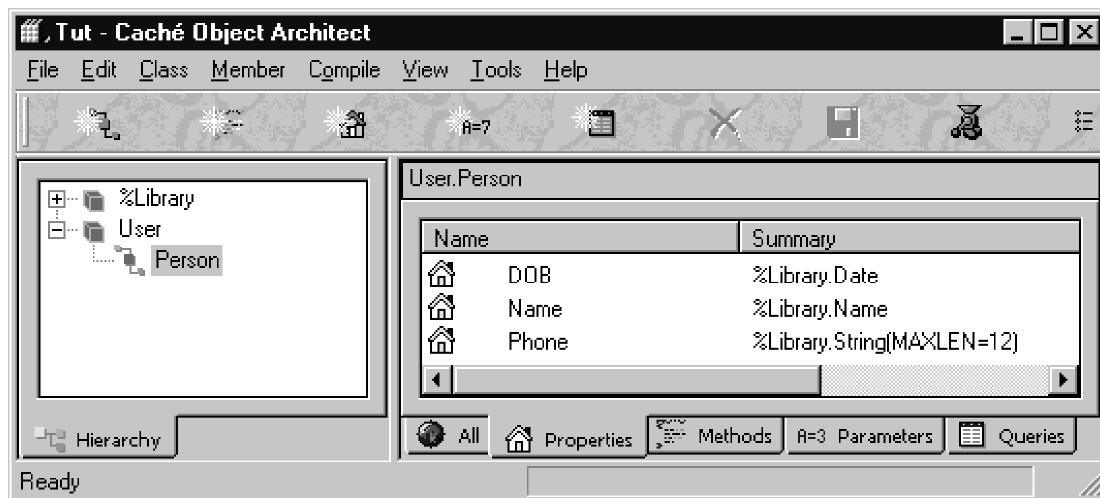


The Caché Program Group

You can also access the GUI tools using the Windows Start button to reach the Caché program group.

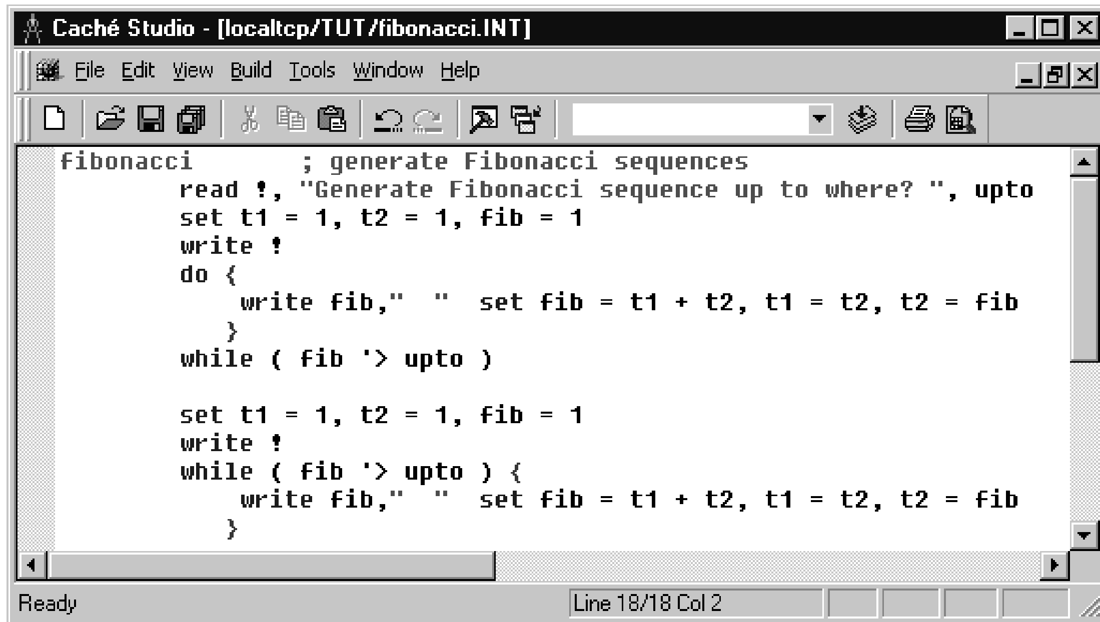
Caché Object Architect

Provides an environment for creating, editing, deleting and compiling classes. Altering classes alters their associated table definitions.



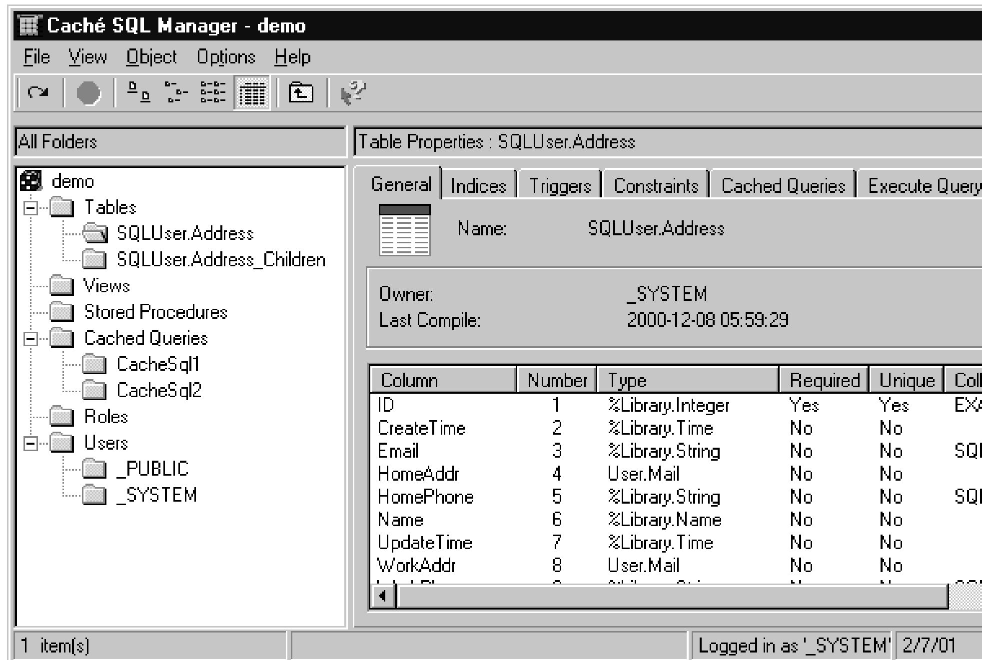
Caché Studio

Allows editing of ObjectScript or HTML.



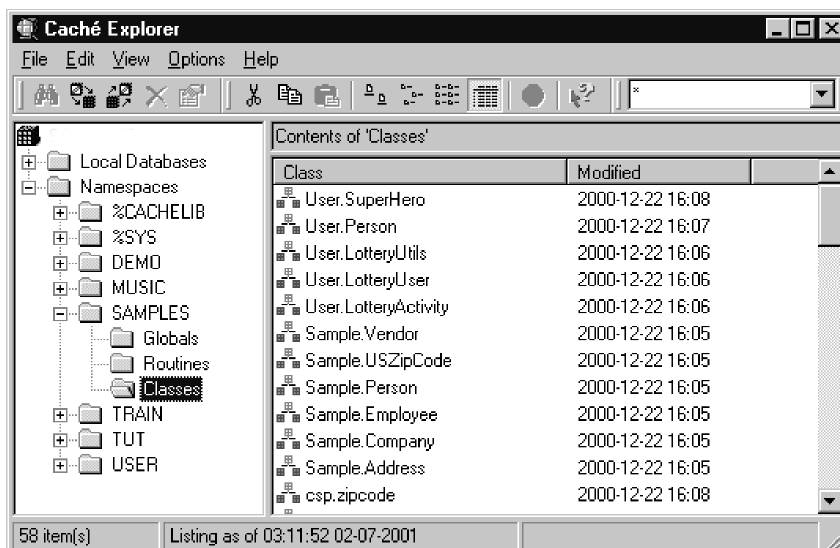
Caché SQL Manager

Provides a relational database view of Caché.
Access the table definitions from here, as well as the data in the tables.
The "Execute Query" tab executes any valid SQL statement on the Caché database.



Caché Explorer

Provides access to class, global and routine management functions.



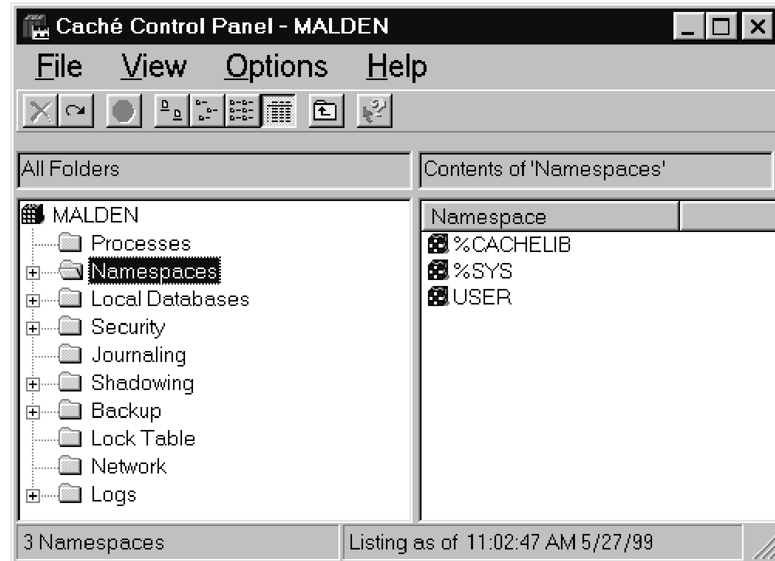
Caché Terminal

Provides direct access to the real-time ObjectScript programming environment.

```
Cache TRM:348
File  Edit  Help

USER>

USER>w $SYSTEM.OBJ.Version()
Cache Objects Version 4.0.818.0
USER>

USER>
```

Caché Control Panel

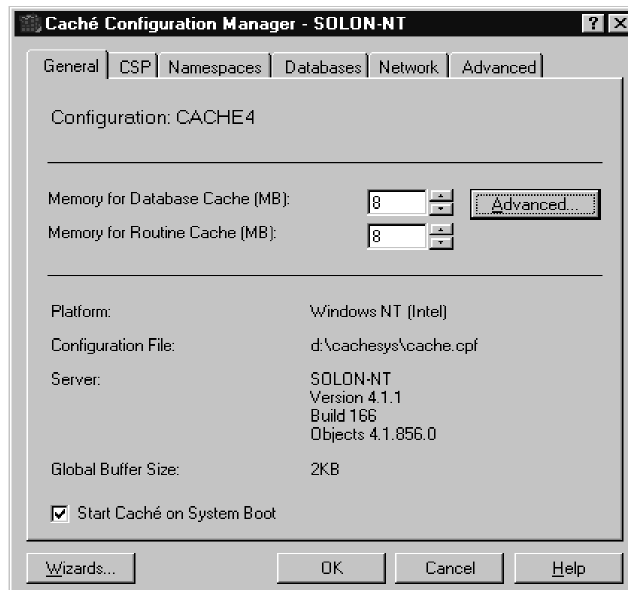
Provides an interface to system management utilities.



Caché Configuration Manager

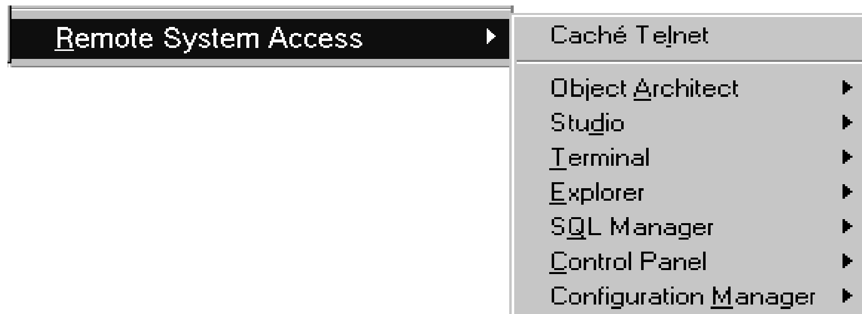
Provides an interface to create and modify namespaces, databases, network connections, and Caché Server Pages applications.

The "Wizards" button also provides access to the License and Printer wizards.



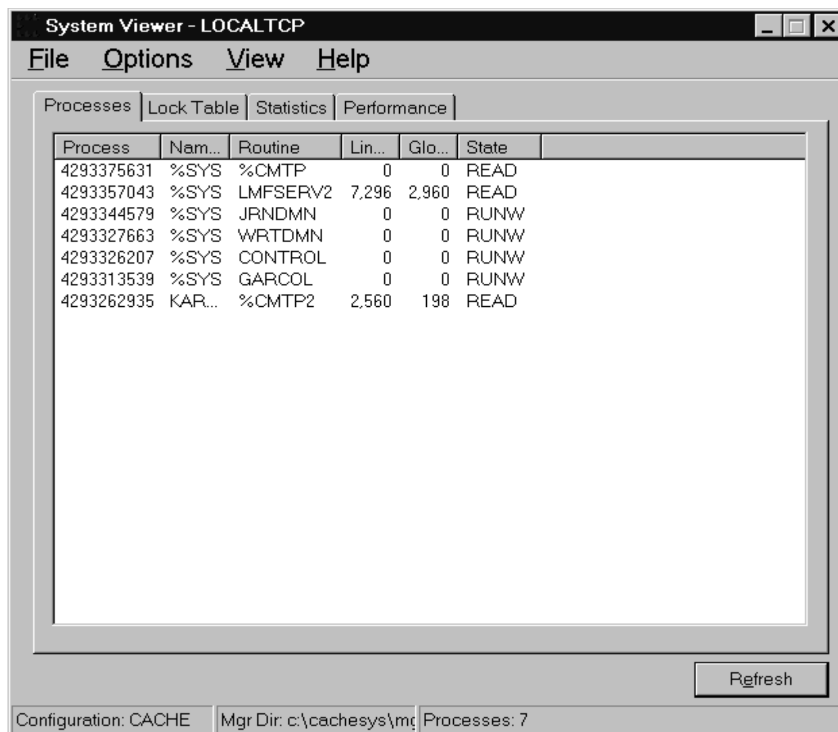
Remote System Access

Using the "Remote System Access" option allows management of multiple systems from a single PC.



The System Viewer

The System Viewer monitors system processes, performance and database locks on the local host.



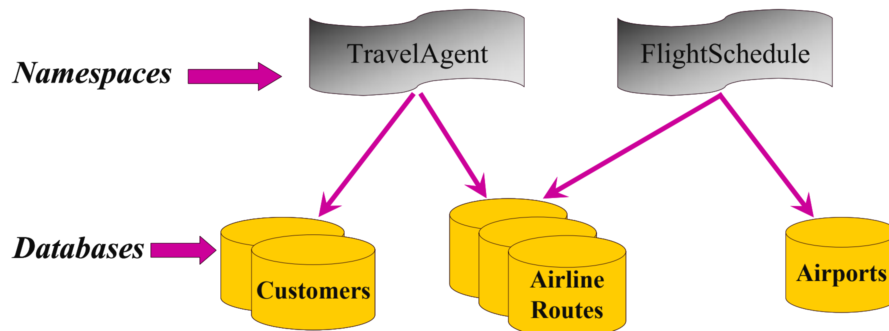
MODULE 2: GETTING STARTED

2 GETTING STARTED

Basic Structure

Data and programs are stored in Caché in Databases, and referred to by Namespaces.

Databases are physical storage locations.
Namespaces are logical references to several Databases.



Databases

Each Caché database represents a cache.dat file somewhere in the file system of the underlying operating system.

A database contains programs (routines) and data (globals).

A database may grow, up to 16GB (32TB in version 4.1) in three ways:

Automatically, in chunks whose size you specify
To a new size you specify
By adding an extent (cache.ext) in another directory, usually on another disk. A database may have up to seven extents.

Routines

Developers usually write either MAC routines or INT routines, but there are four extensions in all:

INC routines contain common code that may be included in MAC routines.
MAC routines contain code, including references to INC routines, Embedded SQL or Embedded HTML. When a developer compiles a MAC routine, Caché first creates an INT routine, followed by an OBJ routine.
INT routines contain ObjectScript code only, and Caché compiles them into OBJ routines.
OBJ routines contain executable code.

Globals

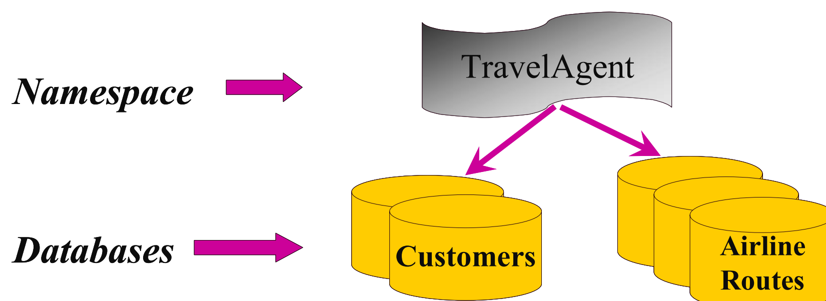
Globals are persistent, multi-dimensional arrays.
The name "global" indicates that potentially all users on a system can access the data.
Multiple dimensions allow globals to contain hierarchical data. For example, for invoices containing transactions:

One global could contain the invoice data, and another could contain the transaction data, or

A single global could contain invoice data at one level, and the transactions for each invoice at another level. With this more typical structure, "joins" are implicit and fast.

Namespaces

All access to Caché databases is via namespaces. This separates the application code from the physical location of the databases.
A namespace is a logical entity that groups several Caché databases into a single unit.
A database can be part of several namespaces.



Changing Namespaces

A Caché process is associated with a particular namespace.
To change namespace from Object Architect or from SQL Manager, select File/Connect.
It will take you back to the Connection dialog.
To change to a different namespace from Terminal, use the ZNSPACE (or ZN) or %CD commands:

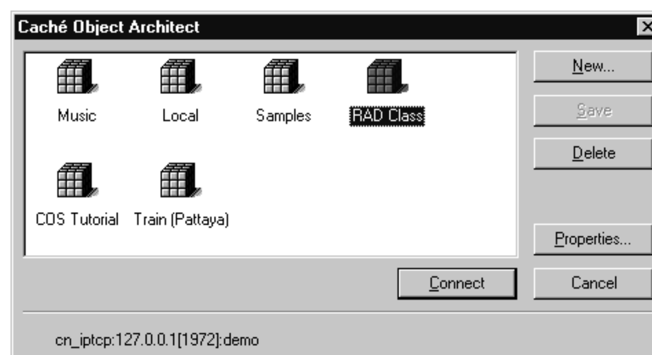
```
USER>zn "%SYS"
%SYS>
%SYS>do ^%CD
Namespace: USER
You're in namespace USER
Default directory is c:\cachesys\mgr\user\
USER>
```

Creating a Namespace

Create a namespace by clicking the Wizards button from the Caché Configuration Manager.
The Wizard prompts for:
a namespace name and
the name of an existing database.
The Wizard also gives you a chance to create a new database.
Databases can be created by clicking the Wizards button from the Caché Configuration Manager.
The Wizard prompts for:
a database name,
whether it is local or remote,
the directory where the database will reside,
the block size of 2kb or 8kb (in version 4.1), and
the initial size in Mb.

Connections

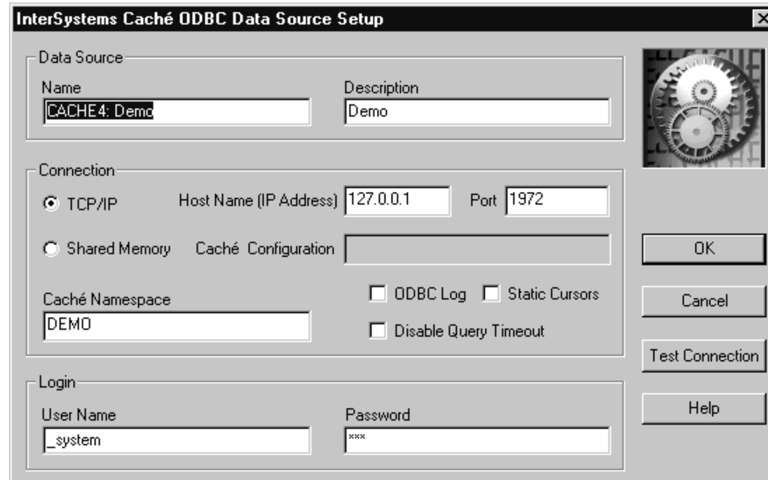
Once you add a new namespace, create a "connection" to it.
Run Caché Object Architect to see the Connection dialog, where you can add a new connection to your namespace.



ODBC Data Source

To provide access to your namespace from other tools such as MS Access via ODBC, you must create an ODBC Data Source for it.

Run "ODBC" from Windows Control Panel, and add a new System DSN that references your namespace.



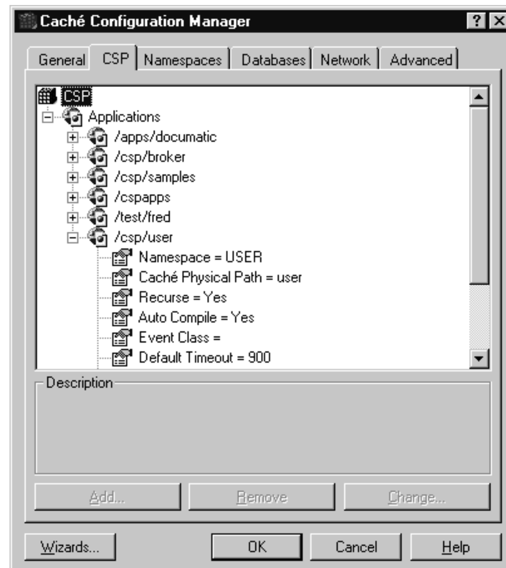
Bi-Directional ODBC

Note that the ODBC access to Caché is bi-directional.
Use MS Access to view and edit data in the Caché database.
Import an MS Access table (its definition and its data) into Caché.
- Use the Access export table feature.
- Caché imports the data and builds the appropriate class.

The Caché SQL Gateway allows Caché to access data in other databases via ODBC.

CSP Application

Caché automatically adds a CSP Application definition for your namespace.
The Physical Path is the directory that holds the .csp source files.
If the directory is the default (c:\cachesys\csp), enter only the sub-directory within that directory.
"user" in the example at right means:
c:\cachesys\csp\user.
Otherwise, enter the full path.



Management

Use Caché Configuration Manager for namespace and database creation.
Use Caché Control Panel for all other management functions, such as creating new extents or doing backups.

Error Handling

Caché displays messages indicating errors on the server.
Find a list of all error messages and their meanings here:
file:///C|/CACHESYS/Docs/err/errindex.html
Sometimes the Terminal prompt will change to something like this: `USER 2d0>`
There is significance to the new prompt, but for now, just enter Q (for Quit) to return the prompt to its normal state.
Refer to Chapter 7 of the Caché ObjectScript Programming Guide for more information.
To debug an error, refer to:
The Debugging section of the ObjectScript tutorial.
Chapter 8 of the Caché ObjectScript Programming Guide.

EXERCISES: MODULE 2

EXERCISES: MODULE 2

In this exercise, you configure a work area (actually several related work areas) for your application, and you configure the Caché environment.

1. Using Windows Explorer, create the directory *d:\mywork* and give it the following subdirectories: *cache*, *csp*, *access* and *VB*.

2. Using Configuration Manager to run the Namespace Wizard, create a new namespace and database, using your first name for both the database and the namespace (*john*, for example). When choosing a location for the database, override *c:\cachesys* with *d:\mywork\cache*. Select the 8k block size for the database. Finish creating the namespace and the database.

3. Continuing in Configuration Manager, go to the CSP tab and find the application that corresponds to your namespace. Change the Caché Physical Path to *d:\mywork\csp*. Click OK and then Activate.

4. Start the Object Architect and create a connection to your new namespace, using the New button. After saving it, connect using your new connection.

5. Set display options (Tools->Options) to *Show inherited members* and *Show members inherited from the system library*. Un-check any other options. Set all four compile options (Compile->Compile Options).

6. Start ODBC Data Sources in Windows Control Panel (Start->Settings->Control Panel). It may or may not be a subfolder of Administrative Tools. Create a new System DSN for your new namespace (System DSN->Add), using the InterSystems ODBC Driver. Give it a name and to specify your namespace. The username is *_system*, and the password is *sys*. Test the connection.

7. Start Caché Control Panel and click *Security* and then *User Accounts*. Double-click the TRM: account and change the namespace to your new namespace. Terminal will now connect to your namespace by default.

ObjectScript Cheat Sheet
Basics

Create a new object: set <oref> = ##class(<pack>.<class>).%New()
Open an existing object: set <oref> = ##class(<pack>.<class>).%OpenId(<id>)
Save an object: set st = <oref>.%Save()
View errors: do $system.OBJ.DisplayError(st)
Close an object: do <oref>.%Close()
Delete an object: do ##class(<package>.<class>).%DeleteId(<id>)
Delete all saved objects: do ##class(<package>.<class>).%KillExtent()
Writing a property: write <oref>.<prop>
Setting a property: set <oref>.<prop> = <value>
Setting an embedded property: set <oref>.<embeddedprop>.<prop> = <value>
Linking two objects: set <oref1>.<prop> = <oref2>

Populate a class: do ##class(<package>.<class>).Populate(<count>)
Close all objects in memory: do $system.OBJ.CloseObjects()
List all objects in memory: do $system.OBJ.ShowObjects() ; "d" for details
Display the contents of an object: do $system.OBJ.Dump(<oref>)

- **Lists**

Create a new list: set <oref>=##class(%Library.ListOfDataTypes).%New()
Insert an element into a list: do <oref>.Insert(<value>)
Display an element of a list: write <oref>.GetAt(<position>)
Display the size of a list: write <oref>.Count()
Clear the elements of a list: do <oref>.Clear()

- **Arrays**

Create a new array: set <oref>=##class(%Library.ArrayOfDataTypes).%New()
Insert an element into an array: do <oref>.SetAt(<value>,<key>)
Display an element of an array: write <oref>.GetAt(<key>)
Display the size of an array: write <oref>.Count()
Clear the elements of an array: do <oref>.Clear()

- **Streams**

Create a new stream: do <oref>.<streamprop>.Write("<text>")
Add text to a stream: do <oref>.<streamprop>.MoveToEnd() and then Write()
Read text from a stream: write <oref>.<streamprop>.Read(.len)
Go to the beginning of a stream: do <oref>.<streamprop>.Rewind()
Clear the contents of a stream: do <oref>.<streamprop>.Clear()
Display the length of a stream: write <oref>.<streamprop>.Size

MODULE 3: TOUR

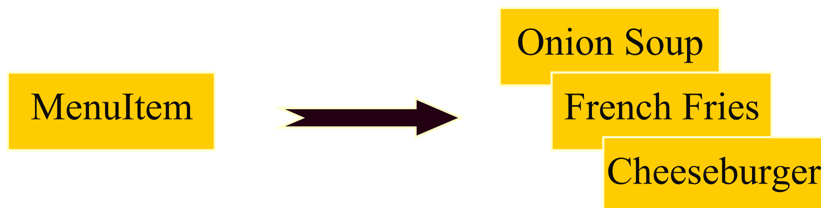
3 TOUR

Example Application

For demonstration, this presentation uses a "Virtual Luncheonette" application.
Hungry people browse the menu, order lunch, and receive notification when their order is ready, all from their office.
One week later, the meal is delivered by a FedEx employee.

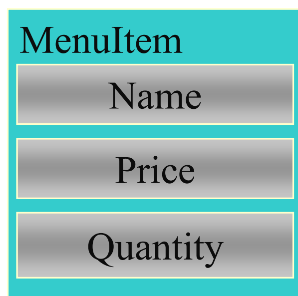
Classes and Objects

A class is a definition of the data together with its application logic.
A class definition consists of:
Properties: definitions of the data elements of the class.
Methods: functions or actions that can be performed on the data.
Queries: SQL specifications for data access.
Parameters: constants for the class.
An object is an instance of a class.



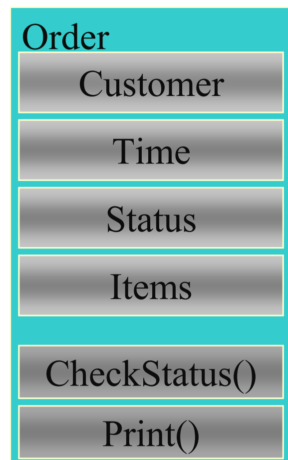
The MenuItem Class

The MenuItem class contains three properties:
Name: the name of the menu item.
Price: the price of the menu item.
Quantity: the number of items currently available.



The Order Class

The Order class has properties to:
identify the customer that placed the order.
track when the order was received.
track the status of the order.
list the items ordered and their cost.
The class also contains some methods:
The CheckStatus() method determines whether or not an order is ready for pick up.
The Print() method prints the list of items and their cost.



Packages

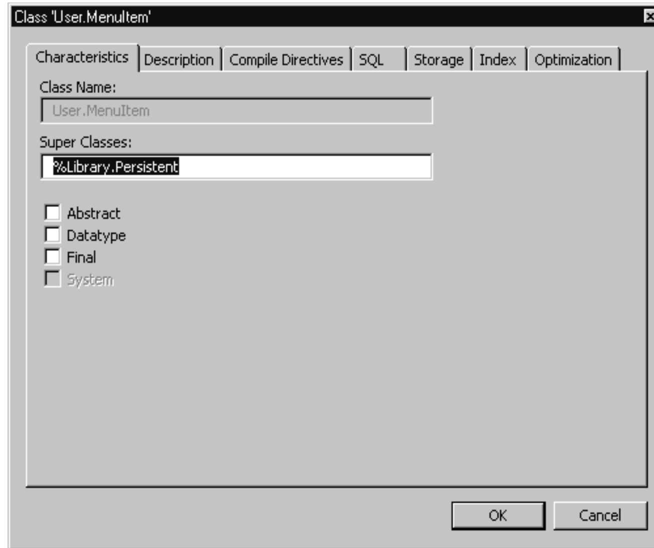
A package (a new feature in version 4.0) is a folder that organizes class definitions within a namespace.
Every package corresponds to an SQL schema, which organizes table definitions in the same way.
The default User package corresponds to the SQLUser schema.
The %Library package contains InterSystems-supplied classes which are available for use in all namespaces.
The %CSP package contains classes used for building Caché Server Pages.
Create a package by naming it when you create its first class. Delete a package by deleting the last class it contains.

Creating a Class

To create a class, use Object Architect's New Class button.
Specify a name and a package, either an existing package or a new one.
Specify what type of class this is.
The most common type, Persistent, indicates that this class defines data that will be stored.
Other types of classes define transient data, or don't define data at all.

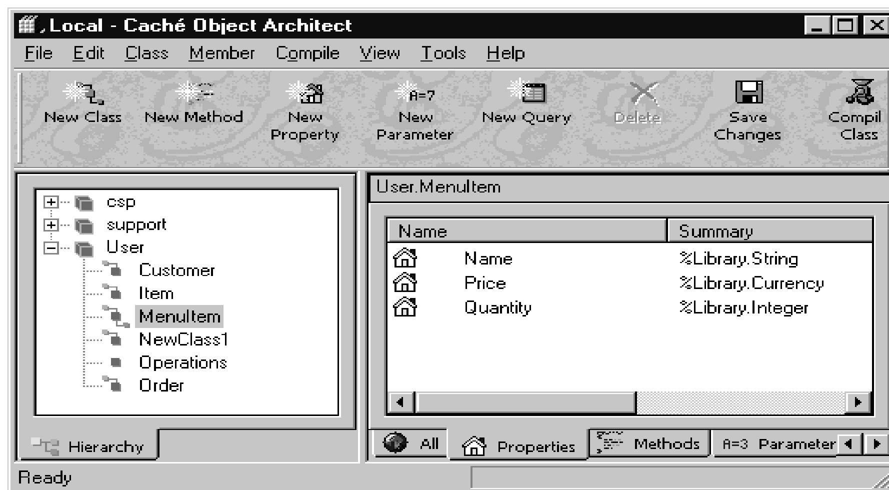
Editing a Class Definition

Once the class is created, add properties, methods, queries, and parameters.
Access other aspects of the class definition by double-clicking the class name in the left-hand pane of the Object Architect.



Properties

Properties define the data stored by the class.
Every property has a name, a type, and an optional set of modifiers and parameters.



The Name Property

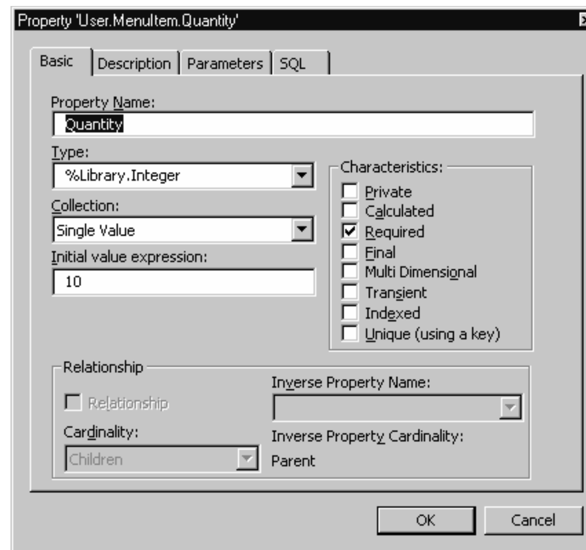
A property type specifies what kind of data is stored there.
In this case the data type is Library.String.
The Required check box indicates that every MenuItem must have a Name.
The Indexed check box tells Caché to create an index of Names.

The Price Property

In this case, %Library.Currency is the data type.
Every property has a set of optional parameters, used to further define the property.
Some parameters are based on data type.
For Price, the maximum and minimum values are specified.

The Quantity Property

Quantity indicates how much of a particular MenuItem is currently available.
An initial (or default) value of 10 means that when a new MenuItem object is created, there are 10 of that item available.



Simple Property Parameters

PATTERN: specify an ObjectScript pattern for string validation. For example: 3n1"-"3n1"-"4n represents a valid phone number.
MINLEN and MAXLEN: specify a range of allowed lengths for strings.
TRUNCATE: specify 0 so that a string longer than MAXLEN will signal an error; specify 1 to truncate that data without error.

More Property Parameters

MINVAL and MAXVAL: specify a range of allowed values (for quantities).
VALUELIST: specify a list of valid values.
DISPLAYLIST: give an optional list of displayable values that corresponds to VALUELIST
FORMAT: an integer, used for numeric values (Integer, Numeric, Date) to specify the acceptable formats. Refer to the Caché ObjectScript Language Reference for specifics:

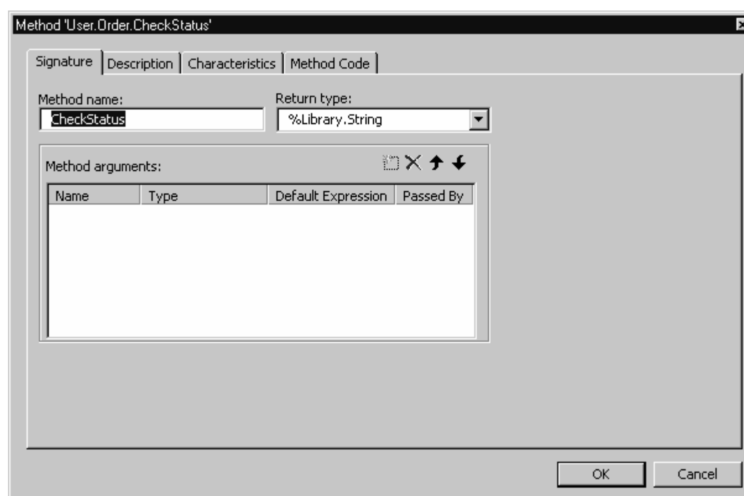
the $ZDATEH function for Date formats
the $FNUMBER function for other numeric formats

Parameters

A property has parameters.
You can't add your own property parameters.
Double click on a property to see the tab that will allow access to the property's parameters.
A class has parameters.
You can add your own class parameters.
To see class parameters, click the class name, and then click the parameters tab at the bottom of the right-hand pane.
Use the New Parameter icon at the top of Object Architect to create a new class parameter.
Double-click any parameter to edit it.

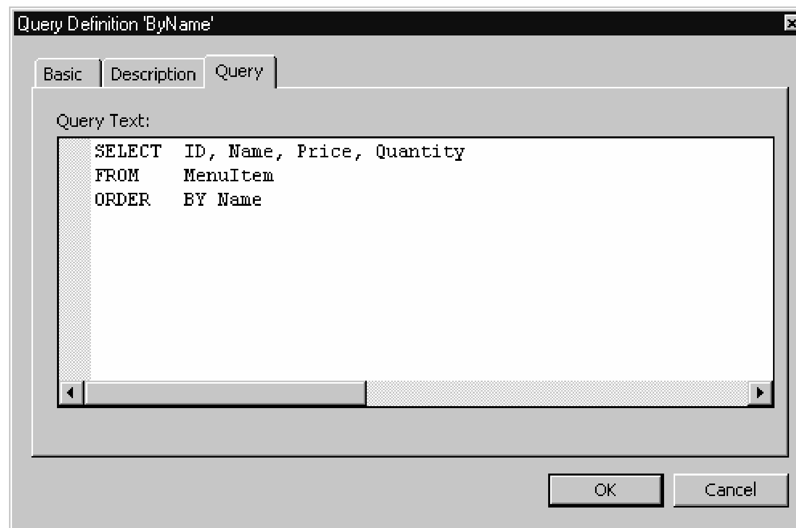
Methods

Caché provides many methods for most of the classes you create.
Inherited classes do common tasks like data storage and retrieval.
Write your own methods using Caché ObjectScript.



Queries

A query is a request for information from a Caché database.
A query is a SQL analog to a method.
A query that references an indexed column in its Where or Order By clause will use the index for efficiency.



Saving and Compiling

After creating a class and adding properties, methods, queries, and parameters, save the class.
When you save a class, Caché generates HTML Class Documentation for it. Access the documentation by right-clicking the class name.
Next, compile the class. During compilation:
Caché processes the class definition and reports any errors.
Caché generates routines (MAC->INT->OBJ). These routines support object access to the data.
Caché also generates table definitions (more meta-data), and routines associated with these definitions. These definitions and routines support SQL access to the data.

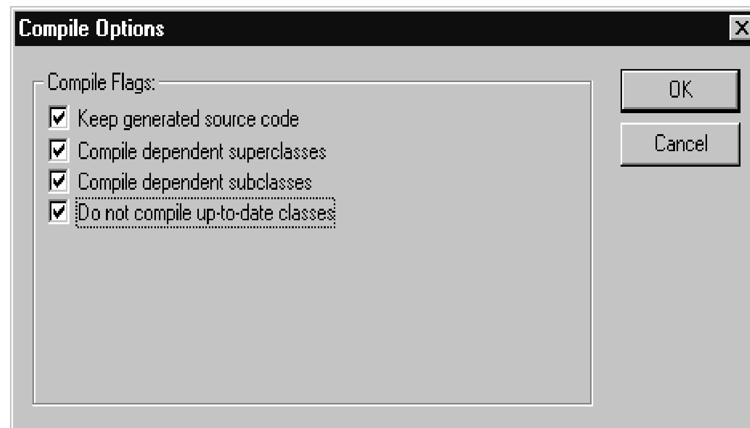
Generated Routines and Globals

Generated routines and globals are named by combining the package name, the class name, and the appropriate suffix.
Routines for the User.MenuItem class:
`User.MenuItem.<#>` support object operations.
`User.MenuItem.<G#>` generate the User.MenuItem.<#> routines.
`User.MenuItem.<T#>` support SQL operations.
Globals for the User.MenuItem class:
`^User.MenuItemD` contains the data.

`^User.MenuItemI` contains indexes.

Compile Options

Compile Options are available via the Compile menu.
The 2nd, 3rd, and 4th options, if checked, cause compilation of a single class to trigger compilation of the class' superclasses and subclasses, if they have been modified since they were last compiled.



Class Definition Language

The Class Definition Language (CDL) is the Caché language for defining classes.
Classes can be defined through the Object Architect or by creating a text file which contains CDL.
Caché can produce the CDL for classes defined in the Object Architect.
CDL is useful for:
moving class definitions between systems.
controlling a development team's access to an application's classes when used as part of a source code control system.

Export/Import of CDL

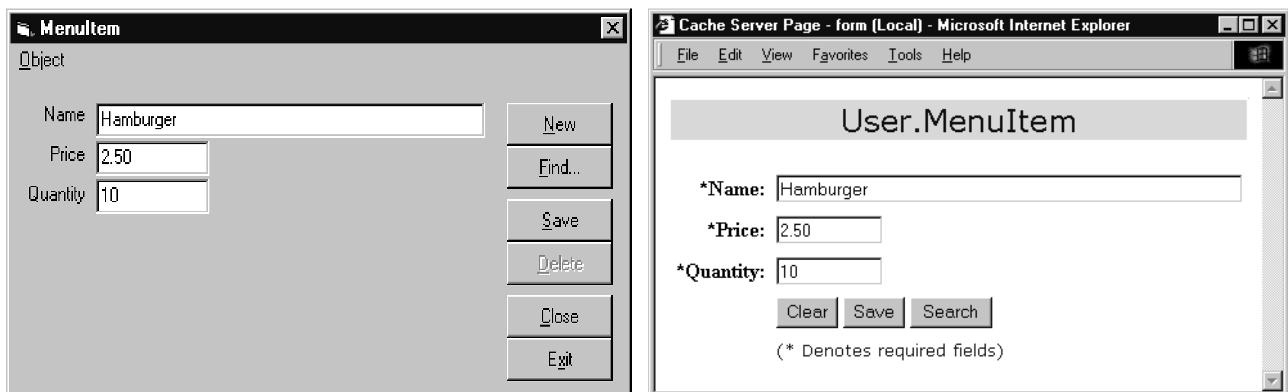
To save the classes for an application:
Use Explorer to select the classes.
Right-click the classes and choose Export.
Use <package>.cdl for the filename.
Move the CDL file to another computer.
To load the application:
Using Object Architect, choose File/Import and select the CDL file.

Creating The User Interface

Create a GUI using VB or CSP.
With VB, the GUI must be distributed to users.
With CSP, a user needs only a browser and the URL.
Caché provides Form Wizards to create simple Visual Basic or Caché Server Page (CSP) forms.
Within the Visual Basic environment, the Form Wizard is available on the Add-Ins menu.
Using Dreamweaver, the Web Form Wizard is available on the Insert menu, by clicking Caché CSP.
Alternatively, run the Web Form Wizard (CacheWebFormWizard.exe) directly from c:\cachesys\bin.
Both forms are fully functional, allowing lookups and editing of Caché data.

Form Wizards

Both Caché Form Wizards prompt for:
a class
properties of the class to display on the form
Here are examples of what they produce:



Using ObjectScript

Write your own code to manipulate objects, using ObjectScript.
Three simple ObjectScript commands:
Set: use to assign a value to a property, and to call a method and capture its returned value
Write: use to display the value of a property or method
Do: use to call a method without capturing its returned value
A dot separates the variable name used to reference an object from a property or method name of that object.
abc.Name is the Name property of the abc object.
abc.Print() is the Print method of the abc object.

Examples of Using Objects

Creating a new MenuItem

```
USER>set mi = ##class(User.MenuItem).%New()
USER>set mi.Name = "Hamburger", mi.Price = 2.50
USER>do mi.%Save()
USER>do mi.%Close()
```

Editing an existing MenuItem

```
USER>set mi = ##class(User.MenuItem).%OpenId(3)
USER>write mi.Name
French Fries
USER>set mi.Quantity = 0
USER>do mi.%Save()
USER>do mi.%Close()
```

%New(), %OpenId(), %Save(), and %Close() are all methods that are provided by Caché.

Syntax Conventions

ObjectScript commands are case-insensitive. They must be followed by a single space.

`##class` is also case-insensitive.

ObjectScript variables, routines, packages, classes, properties, methods, and queries are case-sensitive.

Text strings must be enclosed in double quotes.

Concurrency

Caché offers a means to protect the same object from being edited and/or saved by multiple users at the same time.

The %OpenId() method, and some other methods, have an additional concurrency argument:

```
set cust = ##class(User.Customer).%OpenId(7,4)
```

Concurrency 4 gives one process exclusive access to the object until it is closed.

Concurrency 3 allows multiple processes to open the same object cleanly, but no process can store it until all but one close it.

Concurrency 1 (default) or 2 allow multiple processes to open the same object cleanly, but each process can store it.

Concurrency 0 provides no concurrency control.

Success or Failure

The %OpenId() method returns the empty string (“”) if the ID specified doesn’t exist. If the ID does exist, it returns an integer (an OREF).

%Save() fails if a required property isn’t entered, or a property value is invalid.

The %Save() method returns a status indicating success (1) or failure (not 1).

In order to interpret a failed status, use the %Save() method like this:

```
USER>set st = mi.%Save()
```

To write the error:

```
USER>do $system.OBJ.DisplayError(st)
ERROR #802: Datatype validation failed:
User.MenuItem.Quantity
```

OREFs and IDs

- In the following example, mi is an OREF, and 3 is the ID.

```
USER>set mi = ##class(User.MenuItem).%OpenId(3)
USER>write mi
1
```

- An ID is an unchanging identifier of a persistent object, usually an integer. It is unique within its class.
- An OREF (object reference, or object handle) is a means to access the properties and methods of an in-memory version of an object.
  - Caché assigns an integer to the OREF variable.
- Each time an object is brought into memory, it has the same ID, but may have a different OREF value.

ID

ID is not a property of an object.
Check this out in Object Architect.
ID is a column in every table.
Check this out in SQL Manager.
An object created using %New() has no ID until a %Save().
In a Caché routine, use the method %Id() to find an ID from an OREF.
For example:

```
USER>write cust.%Id()
34
```

OIDs

In the following example, (3,“User.MenuItem”) represents the OID.

```
set mi = ##class(User.MenuItem).%OpenId(3)
```

An ID is unique only within its class.

An OID (object ID) is an unchanging identifier of a persistent object. It is unique throughout an application.

An OID combines the ID with the name of the package and class.

The actual form of an OID is not important here. (3,”User.MenuItem”) is an illustration.

OIDs are mentioned in the documentation of some Caché methods. However, they are rarely used.

The %Open() method requires an OID as one of its arguments.
The %OpenId(), which uses a simple ID, is more commonly used.

ObjectScript: Commands

Every line of ObjectScript begins with a command.
Enter: `set x = a + b`
Not: `x = a + b`
To run a method, use the do command.
Enter: `do person.Method()`
ObjectScript commands have abbreviations.

```
s is set
w is write
d is do
```

For example: `s x = a + b`

ObjectScript: Objects

To create an object:

```
USER>set pers = ##class(User.Person).%New()
```

To enter information:

```
USER>set pers.Name = "Smith,John"
USER>set pers.HomePhone = "555-555-5555"
USER>set pers.DOB = $zdh("5/27/62")
USER>set pers.Age = 40
```

To save the data:

```
USER>set st = pers.%Save()
USER>write st
USER>do $system.OBJ.DisplayError(st) ; if st not = to 1
```

To display the data:

```
USER>write pers.Name
Smith,John
```

ObjectScript: Pattern Examples

Patterns are a series of <count><code/literal> pairs. For example, a `SSN: 3N1"-"2N1"-"4N`
The count can be exact (3) or a range:
1.4 means one to four of the next symbol
Beginning or end of range may be omitted: .1 means none or one
Some valid symbols:
N for Numeral
U for Upper case letter
L for Lower case letter
P for Punctuation
Optional sub-patterns are allowed. For example, a `Zip Code: 5N.1(1"-"4N)`
Use literals instead of symbols where appropriate.
Don't use 1P when you mean 1"-".

EXERCISES: MODULE 3

EXERCISES: MODULE 3

In this exercise, you create a simple class, and use it in a variety of ways (VB, CSP, MS Access, SQL Manager) that demonstrate the key capabilities of Caché.

1. Open Object Architect. Using the New Class button, create a new class, Person, as a persistent class in the namespace you have just created. For reasons that will become apparent later, call the package *Nothing*. Enter a short description. Using the tabs at the bottom of the Architect, notice that your class already has properties, methods, parameters, and one query.

2. Click the Properties tab. Using the New Properties button, add the following properties. Enter short descriptions for each one. Note: in the DOB property below, the MAXVAL=+$h parameter setting means that the maximum date of birth is today; no dates in the future are allowed.

Name	Type	Characteristics/Parameters
Name	%Library.Name	Required
HomePhone	%Library.String	PATERN=3n1"-"3n1"-"4n, MAXLEN=12
WorkPhone	%Library.String	PATERN=3n1"-"3n1"-"4n, MAXLEN=12
Email	%Library.String	
DOB	%Library.Date	FORMAT=5, MAXVAL=+$h

3. Click on the Methods tab at the bottom of the dialog box and note that the %Close(), %New(), %OpenId(), and %Save() are among the methods listed.

4. Use the New Query button to create a query. Name the query *ByName.* There are no parameters, select all fields (including the ID), no conditions, ordered by Name.

5. Save and compile the newly created Person class.

6. Right-click on the Person class and choose *Show Class Documentation*. Look at the automatic class documentation. Also look at %Library.Persistent's documentation. Your Person class has inherited all the functionality of the Persistent class.

7. Start Visual Basic and create a new Standard EXE project. Use the Caché Form Wizard (Add-Ins menu) to create a form based on all the properties in the Person class. Save your project and form, in *d:\mywork\vb*. Run the form using the Start button (triangle pointing right beneath the Diagram menu). Add (New button), lookup (Find button), and edit entries. Try to save an entry without a name, then with a DOB in the future.

8. Start Dreamweaver. Connect to your namespace. Use the CSP Form Wizard (Insert menu->Caché CSP) to create a form based on all the properties of the Person class, except %Id(). Save your form as *person.csp* in *d:\mywork\csp*.
9. Start Internet Explorer and point your browser at http://localhost/csp/<your namespace>/person.csp. Add (Clear button), lookup

(Search button), and edit entries. Try to save an entry without a name, or with a DOB in the future.

10. Start MS Access to create a New Blank Database. Save it in *d:\mywork\access*. Add a New table, and specify it as a Link Table. Use the drop-down *Files of Type* list box to link to *ODBC Databases* and connect to the Machine Data Source you set up for your namespace. Select your table. Open it and view the data you've entered. Enter a new record from here. When using your table, avoid clicking in the ID column.

11. Start SQL Manager. Click on the Tables folder and then the Person table. Look at the table definition built from your class definition. Look at the data by right-clicking on the table name and choosing *Open Table*.

12. Start Terminal and add and edit records. For example:

```
set pers = ##class(Nothing.Person).%New()
set pers.Name = "Smith,John", pers.Email =
                "jsmith@majorcorp.com", etc.
```

Notes: when setting the DOB property directly, set it to a Caché internal date (a number between 50000 and 58000 will do for testing purposes). If you make an error, use the up-arrow on your keyboard to go back to the last line. Use the back-arrow to correct your mistake, then press Enter.

13. Check the table in SQL Manager before and after your save. Try to save an entry without a name.

14. Using Object Architect, right-click on the Person class and choose *Generate CDL*. Save the file to your desktop, and then open it using Notepad. Scroll through it. Note the global names (^Nothing.PersonD) in the Data Location section near the bottom of the file.

15. Start Explorer and select your namespace. Double-click on *Globals.* To view your data in the Caché database, find the ^Nothing.PersonD global and double-click it.

16. Next, click on *Routines,* and note the Nothing.Person routines. Double-click Nothing.Person.1 (MAC) to launch Studio. Use the drop-down list on the toolbar to find the line labeled %Save; scroll so %Save is the top line in the display. Use the View Other button (the magnifying glass at the extreme right of the toolbar) to load the INT version of the routine, find the %Save line again, and scroll so it is the top line. Use View Other to switch back and forth, and notice the differences between the two %Save subroutines.

MODULE 4: CLASSES

4 CLASSES

Object-Oriented Programming

An object-oriented environment is an organized way of developing and maintaining applications.

A class is a description of a data structure, including methods for operating on the data.

An object is an instance of the class, containing data.

The classes have a hierarchical structure, with some classes inheriting from other classes.

If two classes are similar, they should inherit their similarity from the same class, or one should inherit from class from the other.

The purpose of this structure is a system where all specific coding is done once and in only one place.

Encapsulation

Each class definition contains "everything there is to know" (data and actions) about objects of that class.

For the "Virtual Luncheonette" application:

Developers writing other parts of the application use the Order class without knowing how it works.

To change something about how the Order class works, edit the Order class and re-compile it.

There is no need to find all the places Order is used.

Give the Order class to other application developers by providing the CDL file.

Object Oriented Design

To design an application in an object-oriented environment:

Describe your application in words, using nouns and verbs.

Consider each noun as a class. Define the properties of each class.

Consider each verb as a method or query that belongs to a specific class. Methods and queries constitute the "business logic" of the class.

Design the front end to do most of its work by manipulating objects and their properties and calling methods.

From Words to Classes

Words	Classes
"A customer is a person, and an employee is a person."	The Customer and Employee classes **inherit** from the Person class.
"Persons, stores, and orders all have addresses."	The Address class is **embedded** in the Person, Store, and Order classes.
"Most employees work at a store."	The Employee class **references** the Store class.

Inheritance

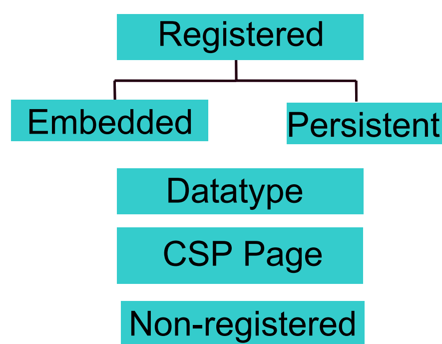
Inheritance enables a class (subclass) to contain methods, properties, parameters, and queries from another class (superclass).
The subclass has its own properties, methods, etc.
It can override properties and methods it inherited.
The following all mean the same thing:
Class B is a subclass of class A
Class B inherits from class A
Class B is derived from class A
Class B extends class A
Class A is a superclass of class B
No class inherits from a class marked as final.

Embedded Classes

Embedded classes are embedded inside persistent classes.
Several persistent classes can re-use an embedded class.
Caché stores the data in an embedded object only when the object containing it is saved.

Kinds of Classes

The diagram at right shows the six different kinds of Caché classes.
Notice that embedded and persistent classes inherit properties and methods from registered classes.



Registered Classes

Registered classes inherit from a system class called %Library.RegisteredObject.
Create a new object from a registered class using %New(), access its properties, and call its methods.
The objects of a registered class cannot be stored.
A registered class does not contain a %Save() method.
Use a registered class for transient objects.
When you're finished with it just %Close() it.
A Print class, with properties Document, Printer, and Copies, and a PrintIt() method is an example of a registered class.

Persistent Classes

A persistent class inherits from %Library.Persistent.
%Library.Persistent inherits from %Library.RegisteredObject.
Create a new object from a persistent class, access its properties, and call its methods.
A persistent class has additional methods to store objects and to open objects.
Store the object, using %Save(), and retrieve it again, using %OpenId().
To be truly persistent, the class **must** have its Persistent Storage option checked.
Double-click the class and click the Storage tab to see this option.

Persistent Object

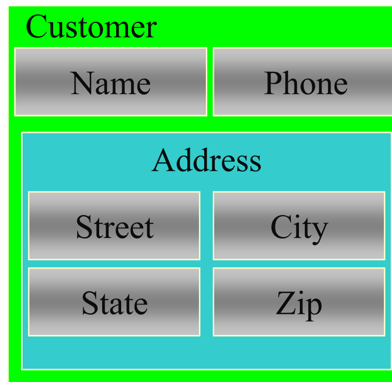
The MenuItem class stores information about each of the available items.

Embedded Classes

An embedded class inherits from %Library.SerialObject
%Library.SerialObject inherits from %Library.RegisteredObject.
Create a new object from an embedded class, access its properties, and call its methods.
An embedded object does not have the %Save() method.
It can be stored only when it's embedded within a persistent object.
To be truly embedded, the class must have its Embedded Storage option checked.
Double-click the class and click the Storage tab to see this option.

Embeddable Object

An embeddable object is stored within a persistent object on disk.
Each Customer has a Name, Phone, and Address.
The embedded Address exists only within the context of a Customer, and is saved only when the Customer is saved.



Storage Options

Caché creates a storage definition for a class when it has a storage option selected.
Without its Persistent option checked, a class that inherits from %Library.Persistent can still be a superclass for other Persistent classes.
It will be a container for properties and methods.
Mark it as Abstract.
The same is true for a class that inherits from %Library.SerialObject without its Embedded option checked.

Abstract Classes

Objects can't be created from an Abstract class.
Any class can be marked as Abstract.
Double-click the class name to edit the class definition.
Click the Abstract option.
An example: a Vehicle class contains properties and methods common to all types of vehicles.
This could be the superclass of Car and Truck classes.
The application code would never create a Vehicle object; only Car and Truck objects.
An abstract class may or may not have a storage definition.
Without storage: cars are stored separately from trucks.
With storage: cars and trucks stored together as vehicles.

Datatype Classes

Caché comes with many datatype classes.
The methods for datatype classes are:
validating the entry before saving it.
converting among different formats as appropriate.
Typical datatype classes include strings, dates, integers, etc.
Caché also allows the developer to create her own datatype classes.
Datatype classes are covered in Module 9.

CSP Page Class

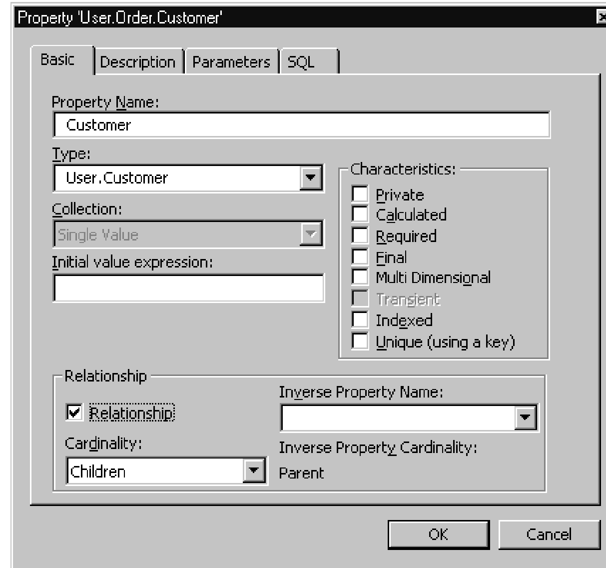
A CSP Page class inherits from %CSP.Page.
Each CSP Page class represents a single Caché Server Page.
Create a CSP Page class either by using Object Architect, or by creating a CSP Page using an HTML editor such as Dreamweaver.
CSP is covered in Module 12.

Non-Registered Classes

A non-registered class doesn't inherit from %Library.RegisteredObject or any of its subclasses.
A non-registered class is a container for
methods (usually), and
properties and parameters (sometimes).
You can't create objects from this class.
A non-registered class is used by other classes.
Other classes can call its methods.
Other classes can inherit from this class.
To create a non-registered class:
create a new class that is Derived, but don't specify a derivation class name.

Referencing Classes

A property of one class can reference another class, via the data type.



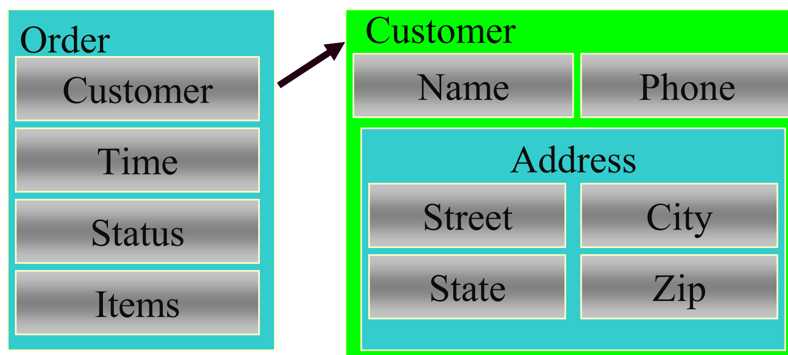
Using the Reference

Our application stores orders on different days from the same customer.
That's why there is a Customer property in the Order class.
Each Order stores a pointer to the Customer that submitted the Order.
Since an Order references a Customer, access the properties of the Customer by chaining the property references:

```
write ord.Customer.Name
```

Linking the Classes

The Type of the Customer property references the Customer class.



The Property Type

The Type field of a property can refer to three kinds of classes:
a datatype class, for simple validation and conversion of the data,
an embedded class, or
a persistent class.
To interpret a property's Type field look at the characteristics of the class to which it refers.

Syntax for Referencing Objects

Linking an Order and a Customer:
```
USER>set ord = ##class(User.Order).%New()
USER>set cust = ##class(User.Customer).%OpenId(4)
USER>set ord.Customer = cust
```

Linking a Customer and an Address:
```
USER>set cust = ##class(User.Customer).%New()
USER>set cust.Name = "Smith,John"
USER>set cust.Address.Street = "10 High St."
```

Or, since Address is embedded:
```
USER>set cust = ##class(User.Customer).%New()
USER>set add = ##class(User.Address).%New()
USER>set cust.Name = "Smith,John"
USER>set add.Street = "10 High St."
USER>set cust.Address = add
```

SQL Projection

Persistent classes are projected as SQL tables upon compilation.
To allow class names and property names that are also SQL reserved words (Order and Date, for example), turn on Delimited Identifiers.
This feature allows you to place quotes around SQL reserved words.
This option is on the Configuration Manager, Advanced tab, SQL section.
SQL is covered in Module 10.

%SYSTEM.OBJ Class

Caché provides some useful methods via the %SYSTEM.OBJ class.
You've already seen DisplayError().
Some of the methods work with objects in local memory, and others work with classes.
See the %SYSTEM.OBJ class for a complete list.
You can call the methods in two ways:
Standard: `do ##class(%SYSTEM.OBJ).DisplayError()`
Shorthand: `do $system.OBJ.DisplayError()`

Using %SYSTEM.OBJ

CloseObjects(): do a %Close() on all open objects.
You've recompiled a class while an object of that class was open in your partition. You can't use %Close() on that object anymore.
Dump(<oref>): show the details of the oref object
ShowObjects(): show the OREF of all open objects
Use "d" as an argument to see a dump of all open objects.
To programmatically access all open objects and their classes, use the ObjectList query.
You'll learn about how to use queries in Module 10.

Object Modeling

Rational Rose (Rational Software Corporation) is the best known tool for creating object models, using UML (Unified Modeling Language).
Use Rose to graphically create an object model.
Rose supports "round-trip engineering," which keeps code and object model in sync.
Rose creates class definitions in many standard programming languages (C++, Java), based on the model.
Rose can import class definitions and update the model.

RoseLink

Caché-provided RoseLink adds Caché commands to the Rose menu. Create classes in Rose and export them to Object Architect, and vice versa.
Caché installs RoseLink if Rose is on your system.
Microsoft Visual Modeler is part of Visual Studio, Enterprise Edition. It is a "mini" version of Rational Rose. To use it with Caché, you must:
Install RoseLink yourself by running c:\cachesys\bin\CacheRoseLink.exe, and modify the Registry as shown on the next slide.

Microsoft Visual Modeler

For those interested in using RoseLink with Visual Modeler, here are the Registry settings:
Add key: [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Visual Modeler\Addins\Cache]
Add the following string values (substitute **your** installation "bin" directory for "c:\cachesys\bin"):

Active	Yes
Company	InterSystems Corporation
InstallDir	c:\cachesys\bin
LanguageAddIn	No
MenuFile	CacheRoseLink.mnu
PropertyFile	CacheRoseLink.pty

Use Tools->Caché to access RoseLink from Visual Modeler.

Nothing Etc.

The rest of the exercises all pertain to an application you'll build for Nothing Etc. Some information about the company.

Mission Statement: "You'll Get Nothing From Us"

Surveys indicate that companies around the world will spend approximately 10% of their budgets on nothing each year.

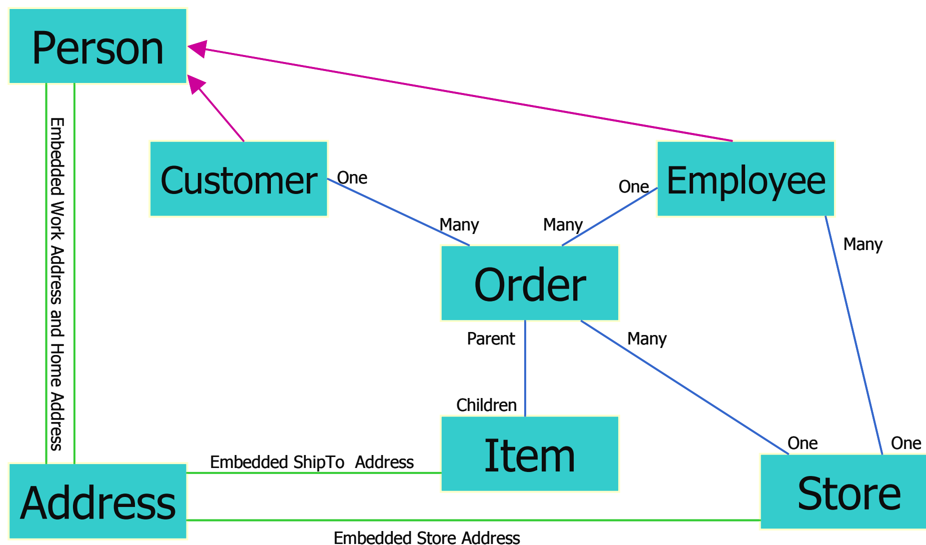
Our application allows our employees at any of our stores to sell nothing to customers in any quantity they desire. Our customers can buy nothing at any time and have it shipped to them anywhere in the world -- free.

We know which employees have made the most money for our company by selling nothing.

Our customers can view their account history at any time.

The following slide shows a modified UML object model for Nothing Etc.

Nothing Object Model



ObjectScript: Date Function

Write a date in the internal format:

```
USER>write $zdateh("1/1/2000")
58074
```

Write a date in various formats:

```
USER>write $zdate(58074)
01/01/2000
USER>write $zdate(58074,2)
01 Jan 2000
USER>write $zdate(58074,5)
Jan 1, 2000
```

Write today's date:

```
USER>write $zdate($horolog)
USER>write $zdate($horolog,5)
USER>write $zd($h)
```

EXERCISES: MODULE 4

EXERCISES: MODULE 4

In this exercise, you build several classes, beginning the process of defining the data for the Nothing application. Two of the classes inherit from the Person class. You use basic characteristics and parameters to define the properties specific to these new classes.

1. In the Person class in the Nothing package, double-click the class name, and check the *Abstract* option. Go to the Storage tab, and un-check *Persistent*. Select and delete the Storage Definition named *Default*.

2. Using Caché Explorer, find the global that holds the Person data (^Nothing.PersonD) and delete it. From now on, the Person class won't contain any data.

3. Create a new embeddable class, Address, in the Nothing package, with the following properties:

Name	Type	Parameters
Street	%Library.String	
City	%Library.String	
State	%Library.String	MINLEN=2,MAXLEN=2, TRUNCATE=0
Zip	%Library.String	PATTERN=5n.1(1”-”4), MAXLEN=10
Country	%Library.String	

4. Add the two following properties to the Person class:

Name	Type
Street	%Library.String
City	%Library.String
State	%Library.String
Zip	%Library.String
Country	%Library.String

5. Using the Save button, save all your new classes.

6. Create a new derived class, Customer, in the Nothing package, as a class derived from Person. Double-click the class name, go to the Storage tab, and check *Persistent*.

7. In the Customer class, add a CustID property of type %Library.String, and check the *Unique* option. Look for the ByName query inherited from Nothing.Person. Double-click on the query and override the query itself to select from Customer instead of from Person.

8. Create a new derived class, Employee, in the Nothing package, as a class derived from Person. Double-click the class name, go to the Storage tab, and check *Persistent*. Add an EmpID property of type %Library.String and check the *Unique* option. Edit the ByName query to select from Employee.

9. Go back into the Person class and delete the ByName query. If you don't do this, Person will not compile because you deleted the Person storage definition.

10. Create a new persistent class, Store, in the Nothing package, with the following properties:

Name	Type	Characteristics
StoreID	%Library.String	Unique
Location	Nothing.Address	

11. Using Configuration Manager, turn on *Delimited Identifiers* (Advanced tab->SQL). Click OK and Activate.

12. Save all your new classes. Compile all of your newly created classes, by using the Compile->Classes menu and the drop-down box instead of the Compile button.

13. Using Terminal, add yourself as a new record to Customer, supplying data for all the fields, including both embedded Addresses. To enter the correct DOB for yourself, set the DOB property equal to $zdh("mm/dd/yy"). Save the record.

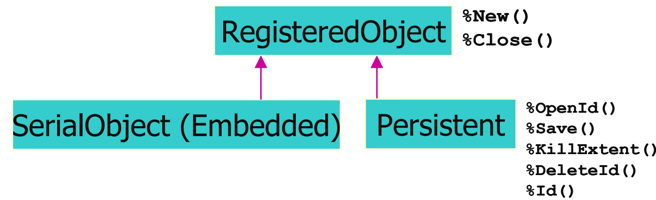
14. Add yourself as a new record to Employee, supplying just the name, and save the record. Use SQL Manager to verify that both records were saved.

MODULE 5: PROPERTIES

5 PROPERTIES

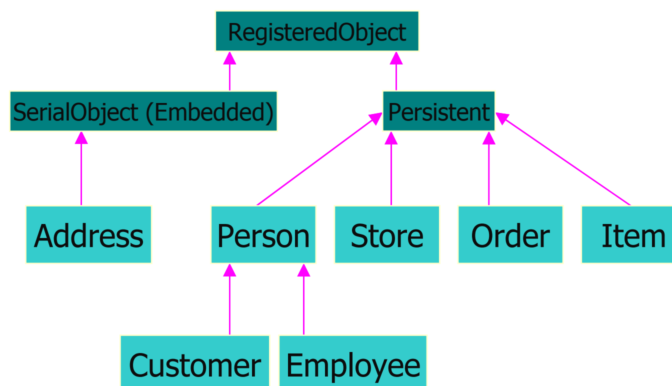
Hierarchy: %Library Classes

Object Classes



Datatype Classes

Hierarchy: Nothing Classes



Properties

Properties define the state of an object. They represent:

- single literal values
- collections of literal values
- streams of characters or binary data
- references to embedded or persistent objects
- relationships between classes

A property that represents a literal value will have, via its data type, methods for validation and translations.

Swizzling

Properties that reference embedded or persistent objects bring them into memory as soon as they are referenced. This practice is referred to as swizzling.

The following line “swizzles” the Customer into memory, via an already-opened Order.

write ord.Customer.Name

The Customer was retrieved without an explicit %OpenId().

Characteristics

A Private property can be used only by methods of its own class.
A Calculated property isn’t stored; it’s value is calculated at run time from other properties.
A Required property must have a value for the object to be stored.
A Final property can’t be overridden by subclasses inheriting it.
A Transient property isn’t stored; it’s only available at run-time.
A Multi-dimensional property is a transient multi-dimensional array.
An Indexed property is indexed for query efficiency.
A Unique property ensures that two objects can’t have the same value for this property.

Collections

The Collection list has the following options, apart from Single Value:

List	(%Library.AbstractList)
Array	(%Library.AbstractArray)
Binary stream	(%Library.BinaryStream)
Character stream	(%Library.CharacterStream)

Information about how to use collection properties is available in the Class Documentation for the classes shown in parentheses above.

Lists and Arrays

Lists and arrays are both used for collections of literal values.
Each item in a collection must have a key.
In a List, elements are identified by position.

For example, if we insert three items into a list, they would be assigned keys 1 through 3. If item 2 were removed, the remaining elements would be assigned keys 1 and 2.

In an Array, elements are identified by a key value that is supplied by the application.

For example, three items in an array, might have keys "A", "X", and "4". If item "X" were removed, the keys for the other items would not change.

A List/Array property can contain up to 32k of data.

Using Lists

Create a list of children:

```
USER>set list = ##class(%Library.ListOfDataTypes).%New()
USER>do list.Insert("Aric")
USER>do list.Insert("Lon")
USER>do list.Insert("Emily")
```

Write the second entry:

```
USER>write list.GetAt(2)
```

Lon

Count the number of gifts needed for the holidays

```
USER>write list.Count()
3
```

Using Arrays

Create an array of children:

```
USER>set array = ##class(%Library.ArrayOfDataTypes).%New()
USER>do array.SetAt("Aric",$zdh("12/16/64"))
USER>do array.SetAt("Lon",$zdh("6/23/67"))
USER>do array.SetAt("Emily",$zdh("4/21/70"))
```

Write the second entry:

```
USER>write $zdh("6/23/67")
46194
USER>write array.GetAt(2)

USER>write array.GetAt(46194)
Lon
USER>write array.Count()
3
```

Lists and Arrays with SQL

A List projects as a single column, with the list items inside it.

An Array projects as a child table containing the array property, with a link back to the parent table.

This is usually a better choice if there are many items in the collection.

Streams

Streams can be used to store large quantities of data, beyond the 32k limit of regular properties.

There are two types of streams, character and binary.

Character streams store large amounts of text, such as a chapter in a book.

Binary streams store large binary objects, such as pictures.

Stream Properties

To create a stream property:
Type: %Library.String
Collection: CharacterStream or BinaryStream.
Caché will change the Type to %Library.Stream.
Store a stream within Caché or in an external file, using the STORAGE and LOCATION property parameters.
STORAGE: Global or File
LOCATION: <global name> or <folder name>
Caché first writes stream data to a temporary location.
When the object that contains the stream is saved, Caché copies the stream to the permanent location.

Using Streams

Streams are embedded objects. Module 4 showed that you can create a **new** embedded object either explicitly or implicitly (using a chained reference).
Unlike other embedded items, to add data to an **empty** stream, create the stream implicitly. Otherwise, Caché stores the data in the temporary location only.
Create a customer and take some notes on it.

```
USER>set cust = ##class(Nothing.Customer).%New()
USER>do cust.Notes.Write("This is my first stored stream.")
USER>set cust.Name = "Doe,Jane"
USER>set st = cust.%Save()
```

Appending to a Stream

Open the object containing the stream.

```
USER>set cu = ##class(User.Customer).%OpenId(34)
```

Create a new stream explicitly.

```
USER>set tempstrm = ##class(%GlobalCharacterStream).%New()
```

Use the CopyFrom() method to copy the stream from the persistent object into the new stream.

```
USER>do tempstrm.CopyFrom(cu.Notes)
```

Go to the end of the new stream and start appending.

```
USER>do tempstrm.MoveToEnd()
USER>do tempstrm.Write("more text")
```

Set the object's stream property to the new stream.

```
USER>set cu.Notes = tempstrm
```

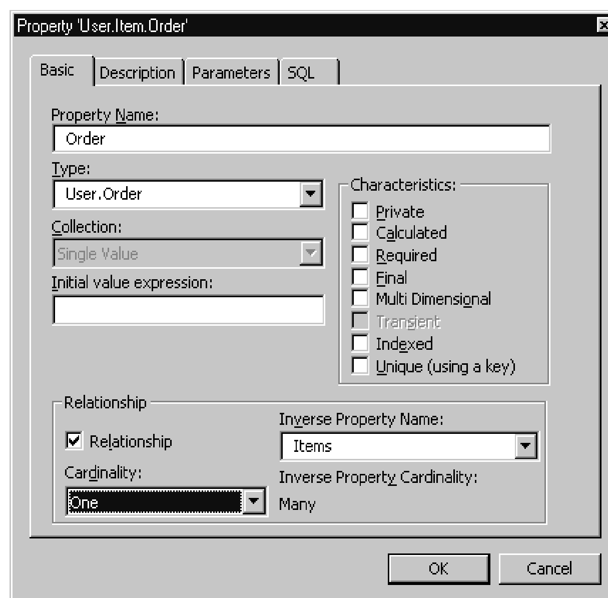
Relationships

Relationships exist between two persistent classes.
A property in one class has a pointer to another class.
Usually, the two classes are different, but a class may have a relationship to itself.
There are two kinds of relationships:
One-to-many (independent)
Parent-to-children (dependent)

Relationships provide referential integrity for deletions on the one/parent side.

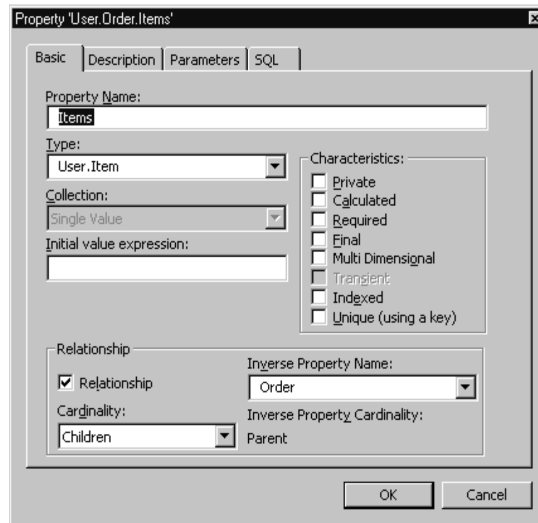
One-To-Many

Use this relationship when:
More than one object in class B (the many) can reference an object of a class A (the one),
Objects in class B may also exist independently without a reference to an object in class A.
An attempt to delete an object of class A will fail as long as it points to at least one object of class B.
To delete an object of class A, delete the references to it from objects of class B.



Parent-To-Children

Use this relationship when:
More than one object in class B (the children) can reference an object of class A (the parent),
Objects in class B may not exist independently of A.
A deletion of an object of class A **will delete all related objects of class B**.
An object in class B with a parent in class A can't be reassigned to a different parent.



Setting Up Relationships

Set up relationships from either the One/Parent side or the Many/Children side.

Add a property that's a relationship to a persistent class and specify its Cardinality. Caché adds the associated property to the other class.

Parent/Children classes compile together.

Relationships are used instead of lists or arrays of objects, because of the referential integrity.

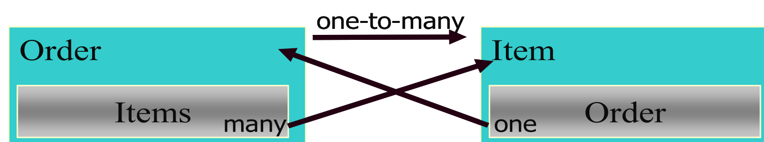
Cardinality

When two classes have a relationship, describe the cardinality in terms of the classes.

"Order has a one-to-many relationship with Item: One Order contains many items."

When setting up the relationship, specify the cardinality in terms of the properties.

"The Items property of the Order class has cardinality Many. The Order property of the Item class has cardinality One."



Using Relationships

Information about relationships is available in the class documentation for %Library.RelationshipObject.

RelationshipObject uses the same method names as AbstractList.

In code, use the relationship from either side. Given an Item it and an Order ord, add the Item to the Order:

```
USER>set it.Order = ord
USER>do ord.Items.Insert(it)
```

Create each new related object explicitly. Connect them as shown above.

Viewing Properties

View properties of related objects using a chained reference.
For example:

```
USER>write it.Order.Date
58700
USER>write ord.Items.Count()
2
USER>write ord.Items.GetAt(1).Price
5.99
```

Parent->Children IDs

IDs of Children in a Parent->Children relationship have the format "a||b":

a = the ID of the parent and
b = the ID of the child within that parent.

For example:

```
USER>write ord.Items.GetAt(1).%Id()
3||1
```

As the argument of the %OpenId method, the ID is in quotes.

```
set it1 = ##class(User.Item).%OpenId("3||4")
```

Using the Children/Many Side

Create a list of objects:

```
USER>set ord = ##class(Nothing.Order).%New()
USER>set item1 = ##class(Nothing.Item).%New()
USER>set item2 = ##class(Nothing.Item).%New()
USER>do ord.Items.Insert(item1)…
```

Writing an object just shows the oref.

```
USER>write ord.Items.GetAt(2)
9 ; just an oref
```

A child has no ID until it is saved.

```
USER>write ord.Items.GetAt(2).%Id()

USER>set st = ord.%Save()

USER>write ord.Items.GetAt(2)
9 ; still just an oref
```

```
USER>write ord.Items.GetAt(2).%Id()
1||2 ; a composite ID
```

Many-to-Many

Many-to-many relationships aren't directly supported, but they can be built using two One to Many Relationships.

Using suppliers and restaurants as an example shows that there are two variations of many-to-many.

Group

There are several groups with members in each class.

Example: a vertical monopoly where several large corporations each own a group of suppliers and a group of restaurants.

Transaction

Each member of each group is an independent agent.

Example: a free market situation where any supplier can engage in a transaction with any restaurant.

Group Example

From the Configuration Manager, select the Network tab.

Click Add… and specify the remote system name and address.

Click OK. A restart will be required to activate the change. For a many-to-many relationship between 2 classes (Supplier and Restaurant), create a third "group" class (MegaCorp) in between Supplier and Restaurant.

MegaCorp has a one-to-many relationship with Supplier, and a one-to-many relationship with Restaurant.

A MegaCorp can't be deleted while it points to any objects in the Supplier or Restaurant classes.

Transaction Example

For a many-to-many relationship between 2 classes (Supplier and Restaurant), create a third class (Transaction) in between Supplier and Restaurant.

Each Transaction is a new relationship.

Supplier has a one-to-many relationship with Transaction, and Restaurant has a one-to-many relationship with Transaction.

Each Transaction represents the reference between one Supplier and one Restaurant.

Neither a Supplier nor a Restaurant can be deleted while it is engaged in a Transaction.

EXERCISES: MODULE 5

EXERCISES: MODULE 5

In this exercise, you enhance your class definitions by using Array, Relationship, and Stream properties.

1. In your Customer class, add a Pets property of type %Library.String. Using the Collection drop-down list, make it an array. Save and compile your class.

2. Using Terminal, create a new array object, using the %New method of the %Library.ArrayOfDataTypes class. Add three or more Pets, real or fictitious, to the array, each keyed by a birth date. Use the SetAt() function from the Cheat Sheet. Then, open the Customer you entered in the last exercise (it probably has ID=1) and assign this array of Pets to your Customer object, using the Pets property.

3. Create a new persistent class, Order, in the Nothing package. Add the following properties:

Name	Type	Characteristics
StoreID	%Library.String	Unique
Location	Nothing.Address	

4. Create a new persistent class, Item, in the Nothing package, with the following properties. Note the comma at the beginning of the VALUELIST parameter; it declares the comma as the delimiter:

Name	Type	Characteristics
StoreID	%Library.String	Unique
Location	Nothing.Address	

5. In the Order class, add the following properties, and then specify that each property is a relationship:

Name	Type	Cardinality	Inverse Property
Items	Nothing.Item	Children	Order
Employee	Nothing.Employee	One	Orders
Customer	Nothing.Customer	One	Orders
Store	Nothing.Store	One	Orders

6. In the Store class, add this property, and then specify that it is a relationship:

Name	Type	Cardinality	Inverse Property
Employess	Nothing.Employee	Many	Store

7. Use Compile->Compile Options and un-check the fourth option. Save and compile all the edited classes.

8. Using Terminal, create one Order and two Items. To assign the Items as children of the Order, use either of the following lines:

```
set it1.Order = ord, it2.Order = ord
do ord.Items.Insert(it1), ord.Items.Insert(it2)
```

9. Save the Order. Create another Employee and one Store, and add the Store to the Employee's record. Save the Employee. Be sure to call %Save() like this: set st = emp.%Save and test the value of *st* to see if the save succeeded. If the save doesn't work, do whatever is necessary to make it work. Then use SQL Manager to verify that everything saved correctly.

10. Use do `##class(Nothing.Order).%DeleteId(1)` to delete the Order and verify that it, along with the two Items, was deleted. Do the same thing for the Store, and verify that it was not deleted. Why can't you delete the Store?

11. Now go back and delete the Employees from the Store. Do this by removing each Employee's Store assignment, and saving the Employee. Or, you can clear all the Employees from the Store and save the Store. Delete the Store again and verify that it was deleted this time.

12. In the Person class, add a Notes property of type %Library.String that is a Character Stream, to contain historical notes about a Person.

13. Override the Notes property in your Customer and Employee classes. Leave the storage as GLOBAL, and set the LOCATION parameter as follows:

Class	**Location**
Customer	^Nothing.CustomerS
Employee	^Nothing.EmployeeS

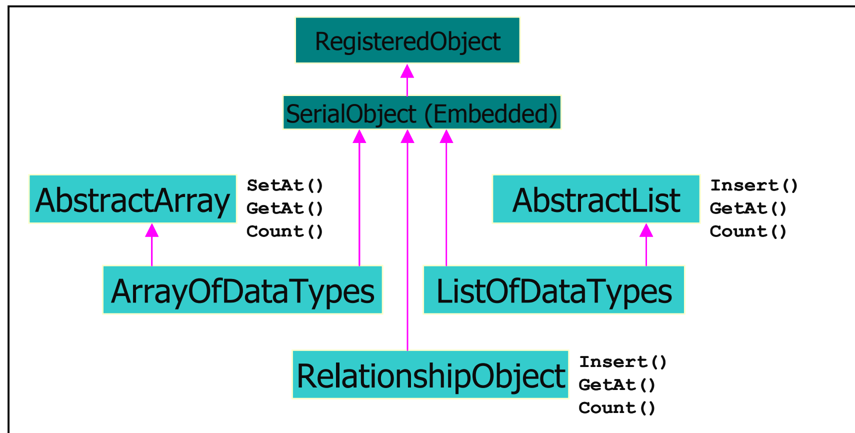
14. Recompile Person. Skim the class documentation for %Library.AbstractStream (click on About Streams).

15. Using Terminal, create a new Customer. Using a chained reference and the Write() method (see the Cheat Sheet), add several lines of Notes to the Customer, and save it. Look at your saved data in the `^Nothing.CustomerS` global.

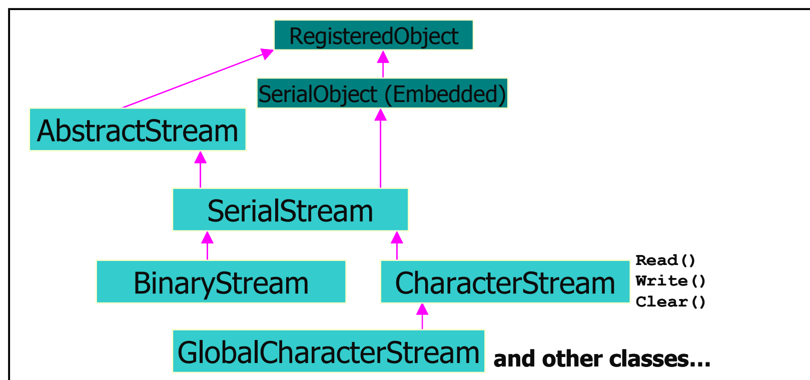
MODULE 6: POPULATE

6 POPULATE

Hierarchy: Collection Classes



Hierarchy: Stream Classes



Populating Classes

Caché provides a simple way to generate random data for classes.
Any class that inherits from %Library.Populate accesses simple methods to generate data.
The methods are in %Library.PopulateUtils.
Using the POPSPEC parameter in each class, associate each property of the class with a method for generating its data.
Call Populate(50) to generate 50 new records.
Use Populate(50,1) to show a success or failure message for each new record.

The POPSPEC Parameter

The POPSPEC parameter is a list of *propname:method()* pairs. For example: City:City(),State:USState().

A property referencing an embedded class should have its own POPSPEC parameter, that uses the PopulateSerial() method.

```
Populate() uses the POPSPEC in the embedded class.
```

For example:

```
.HomeAddr.PopulateSerial().
```

Some properties, based on the property name (Zip), or the datatype (%Library.Name), call the appropriate method even if not referenced via POPSPEC.

More About POPSPEC

For list properties, POPSPEC is *propname:method():maxnum.*

maxnum is the maximum number of items in the list.

For array properties, POPSPEC is *propname:method():maxnum:keymethod()*

keymethod() is the method for generating key values.

For properties referencing other persistent classes, Populate() generates a valid ID pointer to the referenced class.

OnPopulate()

Write an OnPopulate() method for any class that inherits from %Library.Populate.

Populate() calls OnPopulate() just before each newly created object gets saved.

In OnPopulate(), look at the generated properties, and perhaps make some adjustments.

For example, since Name and Gender are simultaneously randomly generated, they might not match. Your OnPopulate() method could simply generate another name, this time based on gender.

OnPopulate() should return a status of success (1). If the returned status is failed (0), that newly generated record will not be saved.

ObjectScript: Conditionals

The If command provides conditional flow control.

```
if <condition> {code}
elseif <condition> {code}
else {code}
```

Save an object in a method. If the status returned is not = to 1, quit and return the status.

```
set st = cust.%Save()
if (st '= 1) {quit st}
```

Post-conditionals are another form of flow control.

```
quit:(st '= 1) st
```

EXERCISES: MODULE 6

EXERCISES: MODULE 6

In this exercise, you learn about the population utilities, and use them to create test data for your classes.

1. Edit your Person class by double-clicking the class name. Add %Library.Populate as a secondary superclass, separating it from the primary superclass with a comma.

2. Override the (empty) POPSPEC parameter in your Customer and Employee classes with the following list of property:method pairs. (Note that Name and DOB will work automatically. Also note that populating the Email property with a company name is intentional; to be used in the exercise for Module 8.):
   `HomePhone:USPhone(),WorkPhone:USPhone(),Email:Company()`

3. Edit your Address class by double-clicking the class name. Add %Library.Populate as a secondary superclass.

4. Override the (empty) POPSPEC parameter in your Address class as follows (Note that Zip will work automatically.):
   `Street:Street(),City:City(),State:USState()`

5. Override the POPSPEC **property parameters** of the Person class as follows (these POPSPECs instruct Caché to find the correct population methods in the Address class):
   HomeAdd .HomeAdd.PopulateSerial()
   WorkAdd .WorkAdd.PopulateSerial()

6. Edit the POPSPEC parameter in your Customer class and add the following property:method():maxnum:keymethod() set to the end (after a comma). This creates an array of up to 3 Pets with names, keyed by a date (the pet's DOB).
   `Pets:FirstName():3:Date()`

7. Edit your Store, Order, and Item classes by double-clicking the class name. For each one, add %Library.Populate as a secondary superclass.

8. Override the POPSPEC **property parameter** of the Store class as follows:
   Location .Location.PopulateSerial()

9. Override the POPSPEC **property parameter** of the Item class as follows:
   ShipTo .ShipTo.PopulateSerial()

10. Save and compile all your classes.

11. In Terminal, use `do ##class(Nothing.Customer).Populate(50)` to create 50 Customers. Do the same thing for Employee, Store, Order, and Item.

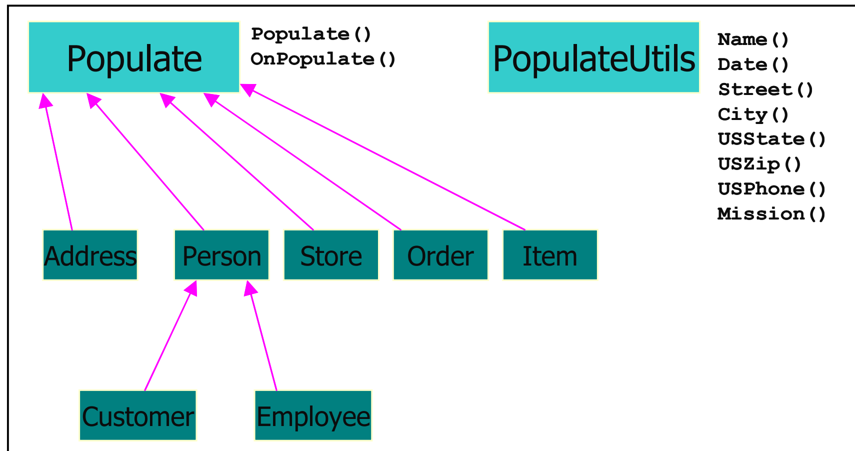
12. Use SQL Manager to look at your data and verify that Populate is working correctly.

13. Start Explorer and select your namespace. Click on *Globals*. Look at all the *D (data) and *I (index) globals. Double-click each one to view it.

MODULE 7: METHODS

7 METHODS

Hierarchy: Population Classes



Methods

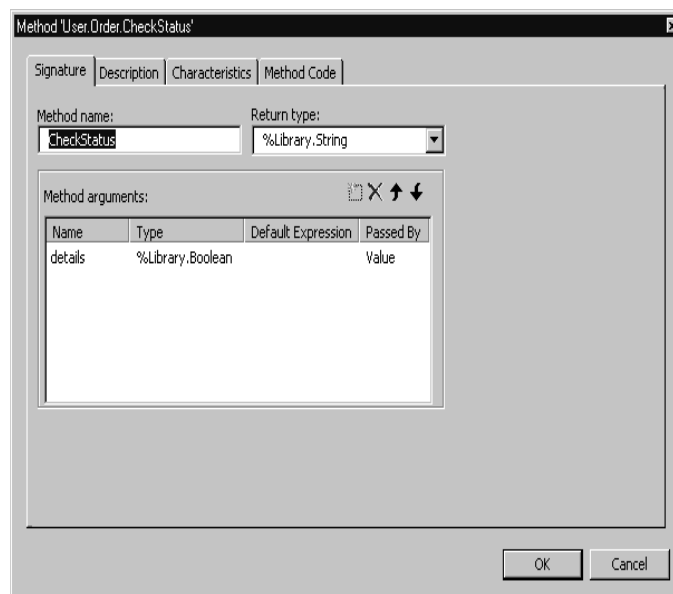
Caché provides many methods for user-defined classes.

The methods are inherited from system classes.

To customize an application, write your own methods with the business logic of your application.

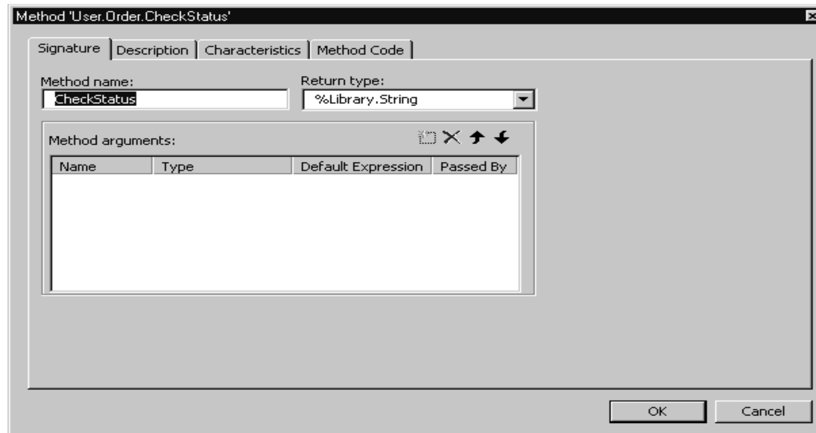
Every method has:

a name,
a return value,
a formal argument specification,
characteristics, and
method code.



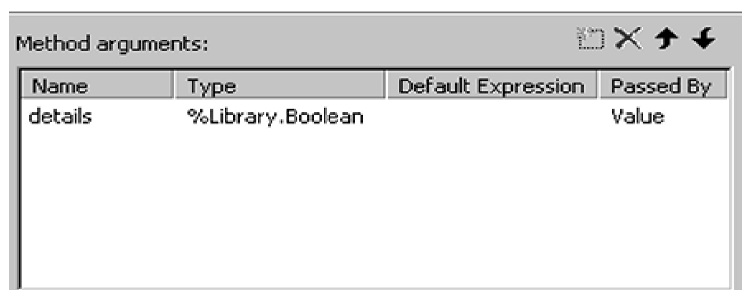
Return Values

Each method's definition specifies its return value.
A method can return any defined data type.
Every method should return a value:
    Either the result of a calculation that the method was written to perform, or
    A value indicating the success or failure of the method.



Arguments

A method's signature specifies:
    the arguments it takes,
    each argument's data type, and
    whether it’s passed by value or by reference.
Methods operate on variables received through their arguments.
A method can take any number of arguments.
By default, arguments are of the %String data type.



Reference vs. Value

Specify that each argument is passed by reference or value to show the syntax for calling the method.
Note that unlike other languages, the caller determines whether an argument is passed by reference or value, not the method code.
Place a period before an argument specified as pass-by-reference.
Pass-by-value: `do obj.Method(a,b)`
Pass-by-reference: `do obj.Method(.a,.b)`

An Example

Consider this simple method:

```
add(a,b)
set a = a + b
quit a
```

If you call the method like this (passing the first argument by **reference**), both sum and x will contain the sum of x and y, because modifying a inside the method modifies x in the caller:

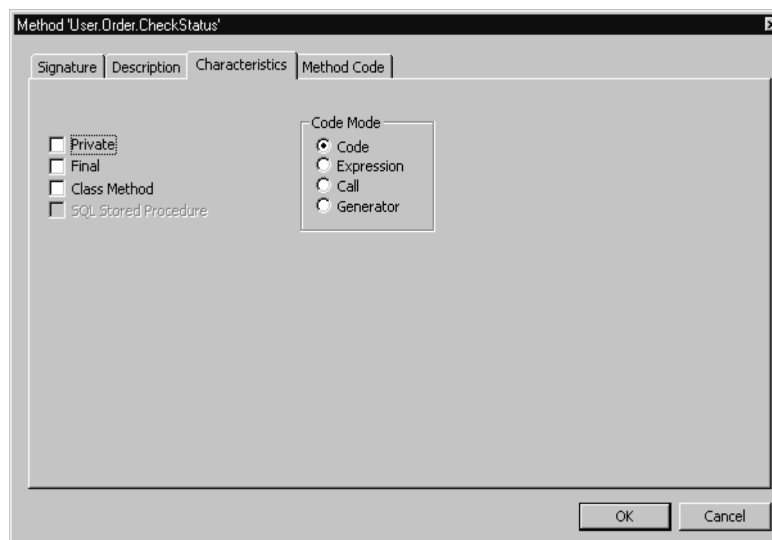
```
set sum = obj.add(.x,y)
```

If you call the method like this (passing the first argument by **value**), only sum will contain the sum of x and y; a is unmodified:

```
set sum = obj.add(x,y)
```

Characteristics

- A Private method can be used only by other methods of its class.
- A Final method can’t be overridden by subclasses inheriting it.
- A method which is not a Class Method is an instance method.



Instance and Class Methods

Always call an instance method for a particular open object.

```
do cust.%Save()  ; save THIS customer!
do cust.%Close() ; close THIS customer!
```

Call a class method when you aren't referencing an open object. Use the `##class` syntax:

```
set cu = ##class(User.Customer).%OpenId(id)
set mi = ##class(User.MenuItem).%New()
```

A class method (usually written using ObjectScript) may be projected as an SQL stored procedure.

Call the stored procedure from a SQL tool such as MS Access.

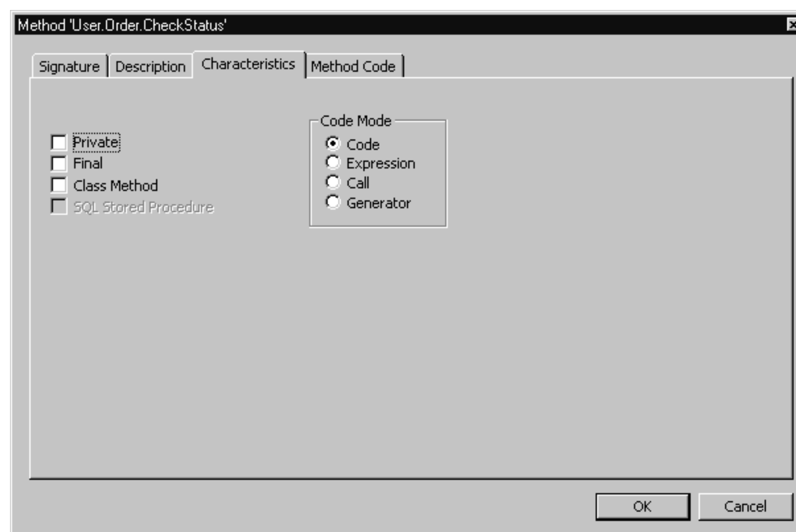
Code Mode

A Code method contains ObjectScript code.

Expression and Call methods are simply alternate ways of specifying simple Code methods:

Expression contains a single ObjectScript expression (argument of a Quit).

Call contains a single routine to run (argument of a Do).



Code Methods

A code method is a method that contains one or more lines of Caché ObjectScript.

The method code can also include embedded SQL and embedded HTML.

Generator Methods

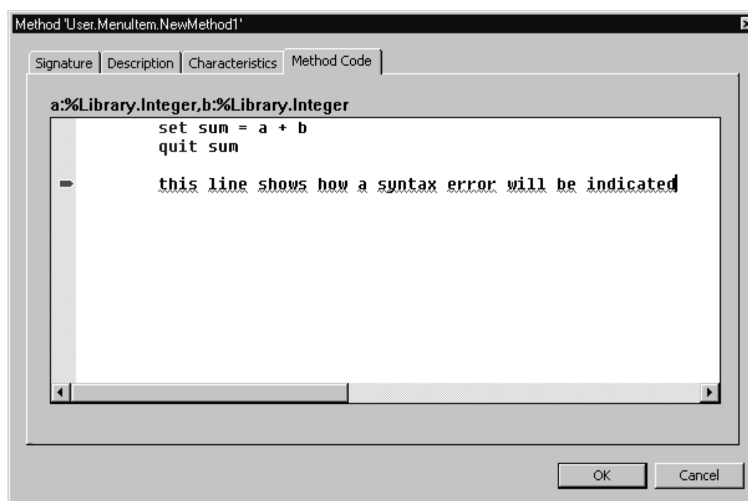
Generator methods are invoked during class compilation to generate a class specific version of the required method.
They do not, by themselves, contain the final executable code.
Generator methods produce high performance, specialized code that is customized to the needs of the inheriting class or property.

A Polymorphism Example

Polymorphism derives from the Greek and means "many forms." Here it refers to a method that has many forms.
The %Save() method (a generator method) is inherited from the Persistent class by many different classes.
Call the %Save() method on any persistent object.
To save a specific object, Caché must know what data the object holds, how to validate it, and where to store it.
When a class is compiled, the generator code generates a %Save() method specific to objects of that class.

The Code Window

For a Code method, each line of code must start with the <tab> character.
To specify the value returned by the Code method, use the Quit command with an argument.
Any syntax errors will be indicated by a red mark.
For expression methods, enter the expression **without** a leading <tab>.



Relative Dot Syntax

The syntax ".." is used to refer to another method or property of the same class.

For example, within the Order class:

`..Time`	the Time property of the current Order instance.
`..Print()`	the Print() method of the Order class.
`..#XYZ`	refers to the XYZ Order class parameter.
`..Customer.Name`	refers to the Name property of the object referred to by the Customer property.

Using Relative Dot Syntax

An instance method has direct access to the property data of its object.
Inside instance methods:

`..<prop>` (another property of this object; this is the **only** way to access private properties)

`..<instance method>` (another method of this class, run on this object)

`..<class method>` (another method of this class, **not** run on this object)

`..#<class parameter>`

Inside class methods:

`..<class method>` (another method of this class)

`..#<class parameter>`

ObjectScript: Formatting

Write to a new line emphatically. Use an exclamation point.

```
USER>write cust.Name, !, cust.Address.City
Doe,John
Boston
```

Make a column using a question mark and a column number.

```
USER>write cust.Name, ?20, cust.Address.City
Doe,John            Boston
```

Add characters within quotes.

```
USER>write cust.Address.City, ", ", cust.Address.State
Boston, MA
```

Use an underscore for concatenation.

```
USER>set line3 = cust.Address.City_", "_cust.Address.State
USER>write line3
Boston, MA
```

ObjectScript: For Loops

For commands have the following syntax:
For <startval>:<incrementval>:<stopval> { <code block> }
Use a number, variable, or method for any of the values.
Omit the third argument only if there is a quit within the loop
The quit will take the processing out of the code block and terminate the loop.

Note: A for loop without a third argument and without a quit is an infinite loop.

Enclose the code block in curly braces.

EXERCISES: MODULE 7

EXERCISES: MODULE 7

In this exercise, you write a simple method to print out any Address in your application, and another method to take care of the grand opening of a new Store.

1. In the Address class, create a Print() **instance** method that prints the Address on 3 lines. The return type should be %Library.Status. Save and compile Address. Test it on all four classes. Use the write command, the relative dot syntax and some formatting characters for the method code.

```
; write the street address
; write the city, state zip
; write the country if it isn't empty
quit 1
```

2. In the Store class, create an Opening() **class** method with two arguments (*emps*, of type %Library.ListOfDataTypes, and *addr*, of type Nothing.Address). Note that you will have to type *ListOfDataTypes* yourself, since the argument dialog box doesn't display this class. The return type should be %Library.Status. Use the Characteristics tab to indicate that this is a class method. Why wasn't this necessary for the Print() method?

The *emps* argument represents a list of Employee ID numbers to be transferred to the new Store. Here is the shell of the method. Replace <number of emps> below with the method that returns the number of Employee ID numbers in the list.

```
; create a new store
      ; set its location to addr
      ; save the store
      ; if the save fails, return the status and quit
            for i=1:1:<number of emps> {
            ; get an id from emps
            ; open the emp with that id
            ; set its store to the new store
            ; save the emp
            ; close the emp }
; close the store
; close the emps and addr
quit
```

3. Save and compile Store. Using Terminal, test the new method. Create two new objects (a list of valid Employee IDs to which you add any number of id numbers, and an Address). Pass these objects as arguments into the Opening() method. Refer to the Cheat Sheet for the syntax for creating a list and inserting values into it.

MODULE 8.1: BUSINESS LOGIC 1

8 BUSINESS LOGIC 1

Hierarchy: First Methods

Address	**Print()**
Store	**Opening()**

Calculated Properties

Calculated properties do not represent stored data.
This data can be quickly calculated from data stored in other properties.
If you have all the elements of a sum, should you store the sum?
Storing it uses valuable disk space.
Calculated properties are always up-to-date.
Calculate age from date of birth.

"Get()" Methods

All stored properties have an associated "Get()" method.
These methods are not visible in Object Architect.
The Name property has a NameGet() method. When you access a property, its Get() method is called. The following two lines do the same thing:

```
write mi.Name
write mi.NameGet()
```

Methods for Calculated Properties

Cache provides no Get() method for a calculated property.
The developer writes one.
The Get() method for any calculated property is an instance method.
It has access to all of the other properties of the current instance, using the ".." syntax.
Users of the calculated property can use it almost like any other property.
But they can't set the property.
If it's possible to reverse the calculation, write a Set() method for a calculated property.
It would allow users of the class to set the property just like any other property.

SQL Computed Fields

Use the SQL tab of a calculated property to specify SQL Computed Field code, so that the property will project to SQL tools like Access.

Caché doesn't allow any extra spaces in this field.

Use {column} to access the value of any column in the current row. It is the SQL equivalent of the .. notation.

Good Coding Practice

The code to perform the actual calculation should exist in one place only.

Write the algorithm into a Calc* class method (CalcAge, for example).

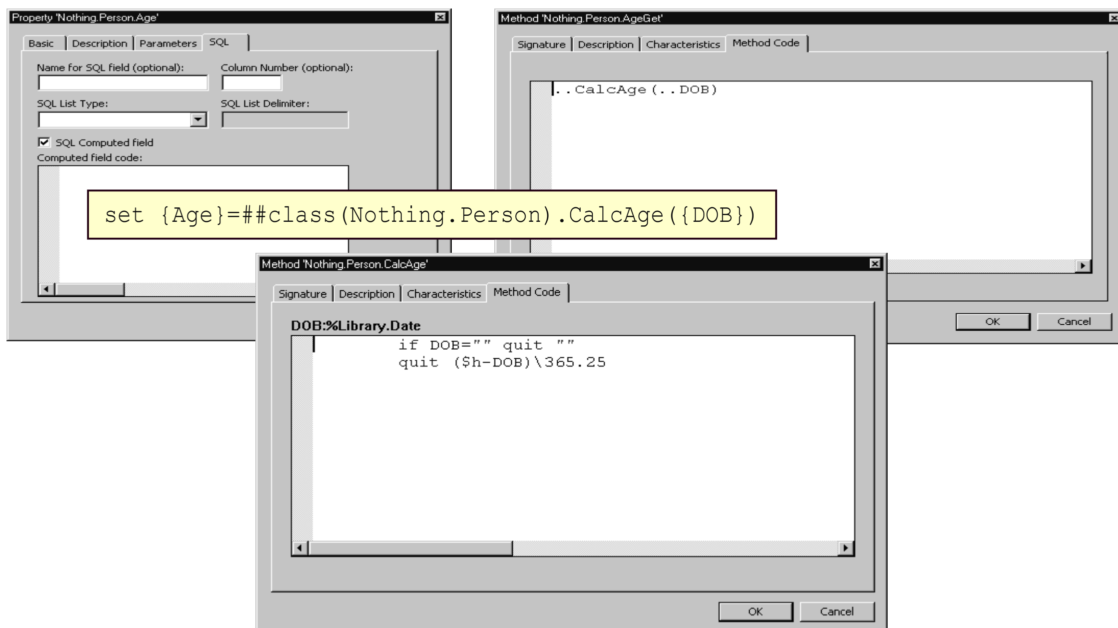
Write the *Get() method in ObjectScript to call the Calc* method for the object side.

This will call the Calc* method using .. syntax.

Double click on the calculated property and select the SQL tab to write the Computed Field code for the SQL side.

This will call the Calc* method using ##class syntax.

Age Example



Callback Methods

Some methods inherited by all registered classes have the ability to call methods (callback methods) that you supply.

Callback methods must match a prescribed signature.

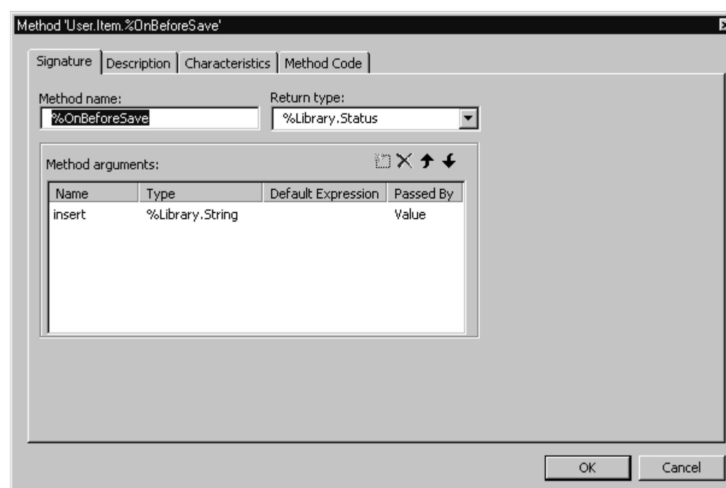
They are **not** called from SQL events (see Triggers in Module 10).

Callback Method Signatures

Method	Return Value	Arguments
%OnNew	return 0 to prevent the %New()	initval (of the object to be created)
%OnOpen	return 0 to prevent the %OpenId()	
%OnValidateObject	return status of customized validation	
%OnBeforeSave	return 0 to prevent the %Save()	insert (=1 for a new record)
%OnAfterSave	return 0 to rollback the %Save();	insert (=1 for a new record)
%OnClose		
%OnDelete (class method)	return 0 to prevent the %DeleteId()	oid (of object to be deleted)

Example: %OnBeforeSave()

The method must have a signature matching the sample to the right.



ObjectScript: String Functions

$Piece takes a string, a delimiter and a number n. It divides the string from left to right, using the delimiter, then returns the nth piece.

```
USER>write $piece("string to parse", " ", 3)
parse
USER>write $p("Cambridge, MA", ",", 1)
Cambridge
```

$Extract returns a substring of a string, given starting and ending positions.

```
USER>write $extract("Add Caché to your app.", 5, 9)
Caché
USER>write $e("Trust your daydreams.", 8, 14)
our day
```

$Justify right justifies by padding with spaces on the left. The third argument specifies number of decimal places.

```
USER>write $justify(3.14159,8,4)
3.1416
USER>write $j(3.14159,8,2)
3.14
USER>write $j(3.14159,0,2)
3.14
```

$Length returns the length of a string

```
USER>write $length("how long is this?")
17
USER>write $l("Thank you.")
10
```

$Translate translates one or more characters in a string into another character.
$tr takes 3 arguments, the string, the old characters, and the replacement characters.

```
USER>w $tr("Cache", "e", "é")
Caché
USER>w $tr("way to go", "wtg", "WTG")
Way To Go
```

$tr also removes one or more characters by leaving the last argument blank.

```
USER>r SSN w !, $t(SSN, "-")
555-55-5555
555555555
```

ObjectScript: Flow Control

The $Case function evaluates its first argument, and returns the string associated with the value.

```
USER>set survivor = 3
USER>write $case(survivor, 1:"Rich", 2:"Kelly", 3:"Rudy",
4:"Sue", :"")
Rudy
```

EXERCISES: MODULE 8.1

EXERCISES: MODULE 8.1

In this exercise, you write several methods: for calculated properties, so that the calculation occurs for both the property and the SQL column; for the Email property, so that it has the proper form, and for the Store class, so that you can delete a Store.

1. In your Person class, create the following two calculated properties:

Name	Type
LastName	%Library.String
Age	%Library.Integer

2. Create a CalcAge() **class** method with one argument (*dob*, of type %Library.Date) that uses $h (today) and the *dob* value to compute the Age. The return type should be %Library.Integer. Note that the back-slash indicates integer division; it divides and returns the integer portion.

```
if dob = "" quit ""
quit ($h - dob) \ 365.25
```

3. Create an AgeGet() **instance** expression method (use the Characteristics tab) that uses relative dot syntax to call CalcAge(), passing the ..DOB property. The return type should be %Library.Integer.

4. Add SQL Compute Code to the Age property that also uses CalcAge(), on {DOB}. Check the SQL Computed Field Code box.

5. Save and compile Person. Using Terminal, test the Age property on an existing Employee and/or Customer. Using SQL Manager, verify that all Employees and Customers have correct Ages.

6. In your Person class, create a LastNameGet() **instance** method that uses $Piece to retrieve the LastName based on the Name. The return type should be %Library.String. Although LastName is a calculated property, one method is sufficient because we will not project this property to the SQL Customer or Employee tables.

```
quit $piece(<name>, ",", 1)
```

7. In your Item class, add 2 new class parameters: ITEMPRICE with value 9.99, and TAXRATE with value .13. Also add the following calculated property:

Name	Type
Price	%Library.Numeric

8. Create a CalcPrice() **class** method with two arguments (*quantity*, of type %Library.Integer, and *size*, of type %Library.String). The return type should be %Library.Numeric. The method uses the *quantity* and *size* values to calculate the Price. Multiply the *quantity* by the ITEMPRICE constant for "Small" nothings and double that for "Large" nothings. Add on the tax, and make sure to round the result to two decimal places.

```
        ; first, set price equal to the quantity multiplied by
    ..#ITEMPRICE
        if size = "Large" { set price = price * 2 }
        ; add the tax, based on ..#TAXRATE, to price
        set price = $j( price, 0, 2) ; round to two decimals
        quit price
```

9. Using Terminal, test the CalcPrice() method, by passing in a *quantity* and *size*. Remember that it's a class method.

10. Create a PriceGet() **instance** expression method that uses CalcPrice(), passing the ..Quantity and ..Size properties. The return type should be %Library.Numeric. Add SQL Compute Code to the Price property that also calls CalcPrice(), using {Quantity} and {Size}. Save and compile Item.

11. To test the new property on an existing Item, go to the SQL manager and find the ID of an existing Item. Make a note of its compound nature. Use the compound ID in quotes in an %OpenId() method, and then write the Price of the Item.
12. In your Order class, add the following calculated property:

Name	**Type**
Total	%Library.Numeric

13. Create a CalcTotal **class** method with one argument (*id*, of type %Library.Integer) that uses an Order's ID value to calculate the Total, by looping through the Items of the Order and summing the Price. The return type should be %Library.Numeric. Make sure to round the result to two decimal places. Here is the shell:

```
; first, set total equal to 0
; open the order, based on id
; if there's no order with that id, return 0
for i = 1:1:<number of items in this order> {
      ; get the price of each item; use a chained reference with
        a GetAt(i) in it
      ; add the price to the total }
; close the order
set total = $j( total, 0, 2) ; round to two decimals
quit total
```

14. Using Terminal, test the CalcTotal() method by passing in the ID of an Order.

15. Create a TotalGet() **instance** expression method that uses CalcTotal(), passing the ..%Id() method. The return type should be %Library.Numeric. Add SQL Compute Code to the Total property that also uses CalcTotal(), with {ID} as an argument. Save and compile Order. Test the new property on an existing Order.

16. In your Person class, override the (empty) OnPopulate() method with code that uses $Piece to build a "real" email address by concatenating the first initial of the name, followed by the LastName property, the @, the first " " (space) piece of the Email (currently a company name), followed by ".com". Save and compile Person. Use Populate() to generate a few more Customers and Employees and verify that the Email property is being set correctly. Here is the shell:

```
set company = $p( $p(<email>, " "), ".com")
; this      also removes any .coms
set <email> = $e( $p(<name>, ",", 2), 1)
    _ <lastname>  _ "@"
    _ company _ ".com"
quit 1
```

17. Create an %OnDelete() **class** method for your Store class. Using the signature given on the callback slides, the first argument will be the OID of the object to be deleted, and a returned value of 0 will prevent the deletion. The return type should be %Library.Status. The oid argument should be of type %Library.ObjectIdentity (note that you will have to type *ObjectIdentity* yourself). The OID value will be supplied by %DeleteId() when it calls the %OnDelete method. The callback method removes (unlinks) all the Employees and Orders from the Store being deleted. Here is the shell:

```
new id ; this is necessary for now
       ; notice that you will not actually use id in the rest of the
method
; open the store with the oid, using %Open(oid) instead of
%OpenId(id)
do store.Employees.Clear() ; unlink this store's employees
do store.Orders.Clear() ; unlink this store's orders
; save the store, returning a 0 if the store does not save properly
; close the store
quit 1
```

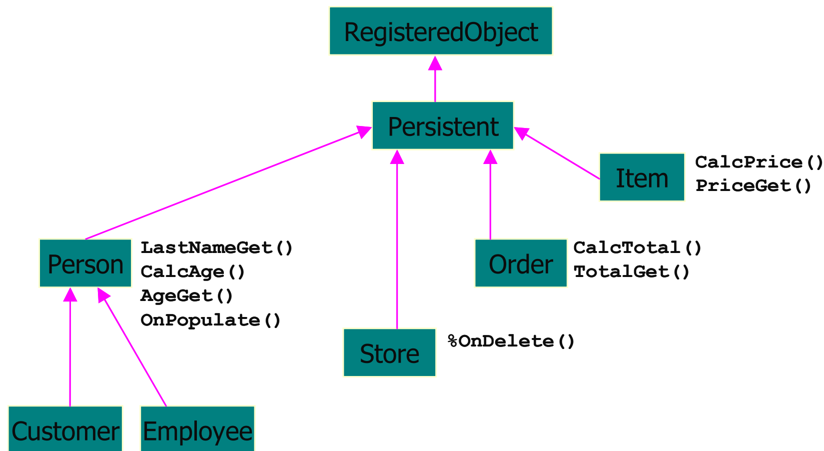
18. Using SQL Manager, look in the Employee table to find a Store ID linked to at least one Employee. Using Terminal, test your %OnDelete() method by calling %DeleteId() on that Store. Unlike Exercise 5, the deletion succeeds.

19. Names in the database are stored as *Last,First*. As an added challenge, either because you are facile in your use of ObjectScript or because you need practice with it, write a new PrintHome() instance method in the Person class to display the name and Home Address of a Person, using the conventional *First Last* format for name. Use the $piece function to reformat the name, then call Print() on the HomeAdd property.

MODULE 8.2: BUSINESS LOGIC 2

8.2 BUSINESS LOGIC 2

Hierarchy: More Methods

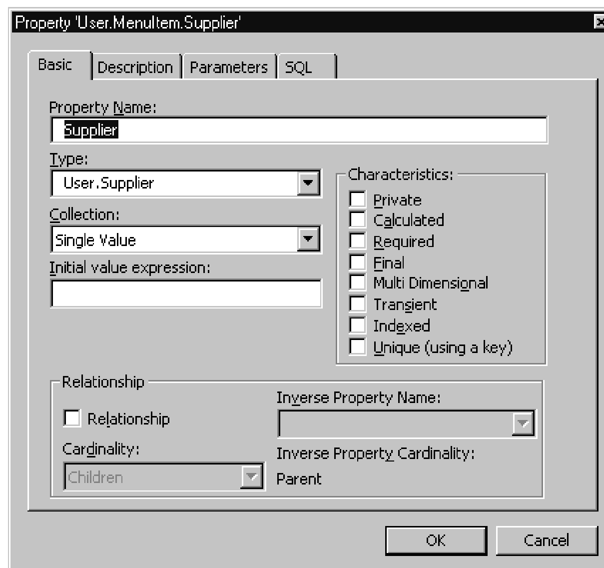


Multi-Referencing

Sometimes, one class needs a property that can refer to one of a set of multiple, related classes.

For example, each MenuItem comes from a supplier. The supplier can be:

a chef at the luncheonette,
a local vendor, or
a national vendor.



From Words to Classes

Although the chefs, local vendors, and national vendors all supply food, the Luncheonette application requires different data about each group, and ordering from each group is different.

The Supplier class contains properties common to all suppliers. It has these subclasses:

Chef
LocalVendor
NationalVendor

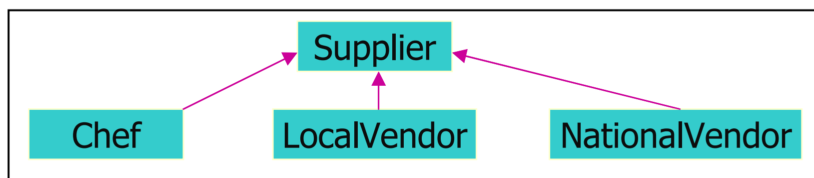
Supplier Object Model

Chef, LocalVendor, and NationalVendor all inherit from Supplier.

Supplier contains properties such as Name and Address, common to all the subclasses.

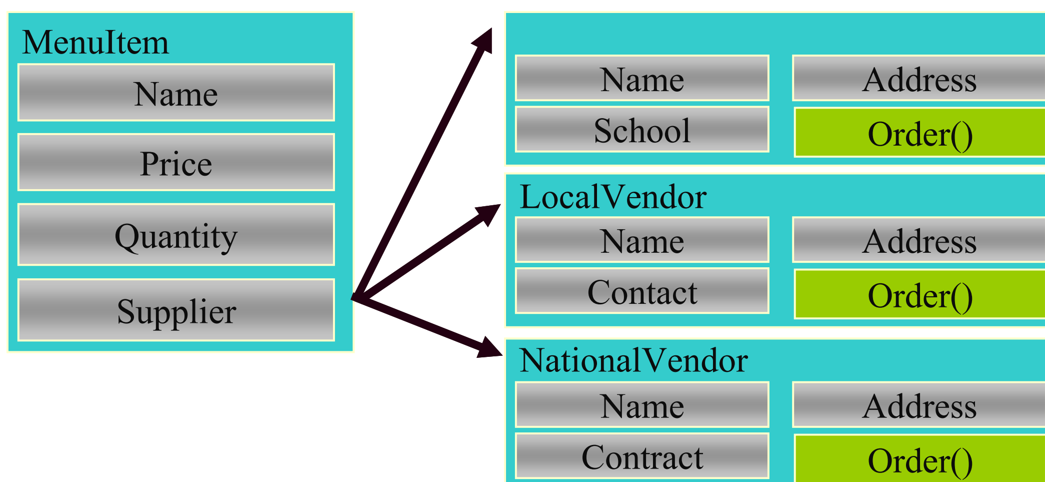
Supplier contains a method, Order(), common to all the subclasses.

The subclasses also contain class-specific properties, such as NationalVendor.Contract.



Using the Multi-Reference

Even though the Supplier property's type is User.Supplier, a MenuItem directly refers to a supplier subclass.



Syntax for Multi-Referencing

The syntax for using a multi-reference is the same as for any reference.
Linking a MenuItem and a Chef:

```
USER>set mi = ##class(User.MenuItem).%OpenId(4)
USER>set chef = ##class(User.Chef).%OpenId(9)
USER>set mi.Supplier = chef
USER>write mi.Supplier.School ; only for a chef!
```

Culinary Institute
All three subclasses have an Order() method. Call it like this:
USER>do mi.Supplier.Order()
Depending on the subclass referenced by the Supplier property, Caché calls the correct Order() method.
This is an example of polymorphic dispatch.

Defining a Multi-Reference

Create a persistent superclass (marked as abstract) with several persistent subclasses.
Add properties and methods to the superclass that are common among all the subclasses.
Where necessary, override the methods in the subclasses, coding specific behavior for each subclass.
The information on the Signature tab of each method overridden in a subclass must "match" the superclass method signature. This means:
The Return Type must match.
The subclass method must have at least as many Arguments as the superclass method; it may have more.

EXERCISES: MODULE 8.2

EXERCISES: MODULE 8.2

In this exercise, you create a set of Payment classes to handle different kinds of payments.

1. Create a new persistent class, Payment, in the Nothing package, with the following properties:

Name	Type
Number	%Library.String
RecvDate	%Library.Date

2. Edit your Payment class by double-clicking the class name. Add %Library.Populate as a secondary superclass, separating it from the primary superclass with a comma. Mark the class as Abstract, but leave the storage definition as is.

3. Create a Process() instance method with one argument (*amount*, of type %Library.Numeric). The return type should be %Library.Status. Caché requires that the method have at least one line of code: `quit`

4. Create three classes Derived from Payment: Check, CreditCard, and PurchaseOrder. They all inherit the two Payment properties, and have additional properties as listed below:

Check properties

Name	Type	Parameters
Bank	%Library.String	VALUELIST=,Fleet,Chase,Citibank,Bank One,First Union

CreditCard properties

Name	Type	Parameters
Type	%Library.String	VALUELIST=,Visa,MC,AmEx,Discover,Diners
ExpDate	%Library.Date	

PurchaseOrder properties

Name	Type
Company	%Library.String
DueDate	%Library.Date

5. In each subclass, override the Process() method. Each method should have one write statement, appropriate to the class, something like this:

```
write "Processing ", ..Bank, " check # ", ..Number,  "payment",
    !, "For $", amount
quit 1
```

For the CreditCard class, print the Type and Number. For the PurchaseOrder, print the Company and Number.

6. In the Order class, add this property:

Name	Type
Payment	Nothing.Payment

7. Compile Order and Payment.

8. Using Terminal, open two existing Orders. Create a new PurchaseOrder object, and a new CreditCard object. Assign the PurchaseOrder a Company and a Number, and link the first Order to the Purchase Order. Use the syntax

```
set pay1 = ##class(Nothing.PurchaseOrder).%New()
set ord1.Payment = pay1
```

Assign the CreditCard a Type and a Number, and link the second Order to the CreditCard. In both cases, use the Payment property to make the link. Save all the objects.

9. Call the Process() method on the Payment property of each open Order, passing the Order.Total as the argument.

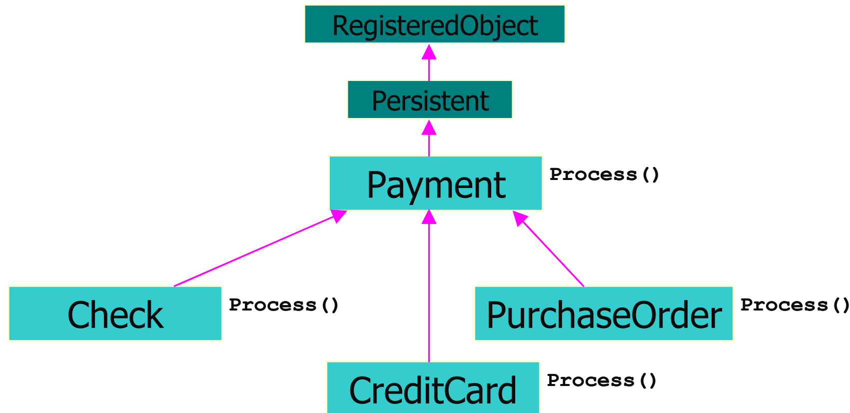
```
do ord1.Payment.Process(ord1.Total)
```

Is Caché processing the Orders correctly?

MODULE 9: NON-REGISTERED CLASSES

9 NON-REGISTERED CLASSES

Hierarchy: Payment Classes

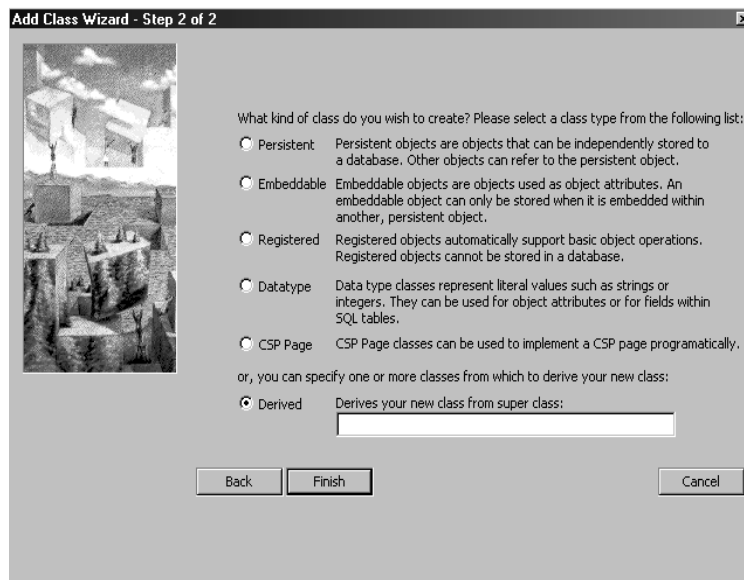


Non-Registered Classes

A non-registered class doesn't inherit from %Library.RegisteredObject or any of its subclasses.

You can't create or save objects using a non-registered class. They are usually containers for class methods.

To make a non-registered class, select Derived, but do not name the superclass.



Datatype Classes

A datatype class is a special kind of non-registered class. It's a container for methods that:

- manage the conversion of literal data between its stored (on disk), logical (in memory), display, and ODBC formats.
- provide validation for literal data values.
- provide for SQL, ODBC, ActiveX, and Java interoperability.

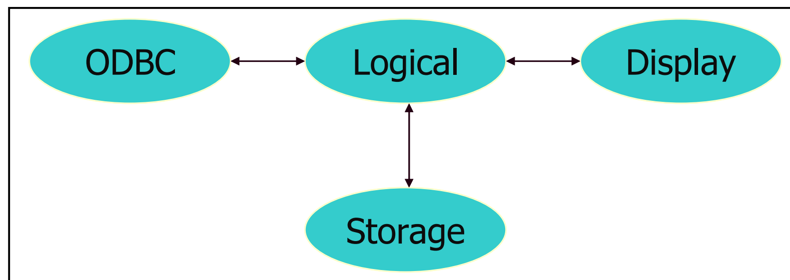
Caché comes with many datatype classes for typical kinds of data: strings, dates, integers, etc.

Caché also allows you to customize datatypes.

- Create your own datatype classes.
- Extend or customize existing classes by using datatype class parameters.

Conversion

You can use datatype methods to convert among the following four formats:



Datatype Methods

Datatype methods have standard names and signatures.

- IsValidDT(): contains code that returns True if the data is valid, False otherwise.
- DisplayToLogical(): contains code that returns the result of converting a displayed value into a different, logical (in memory) value.
- LogicalToDisplay(): contains code that returns the result of converting a logical value into a different value for display.

The other methods are: LogicalToStorage(), StorageToLogical(), OdbcToLogical(), LogicalToOdbc().

Example: %Library.Date

When a user-entered date is saved:

- Caché validates it,
- converts it to an internal format, and
- stores it.

The internal format is an integer which counts days.

- 1 is January 1, 1841.

2980013 is December 31, 9999.

The internal format allows easier comparisons and simplifies arithmetic using dates.

For display, convert dates to a person-friendly format, regardless of how they are originally entered.

Calling Datatype Methods

IsValidDT() is called automatically when a user tries to %Save an object. It can also be called directly.

DisplayToLogical() and LogicalToDisplay() aren't called automatically. Call them as needed.

If the developer doesn't call the date formatting methods, either

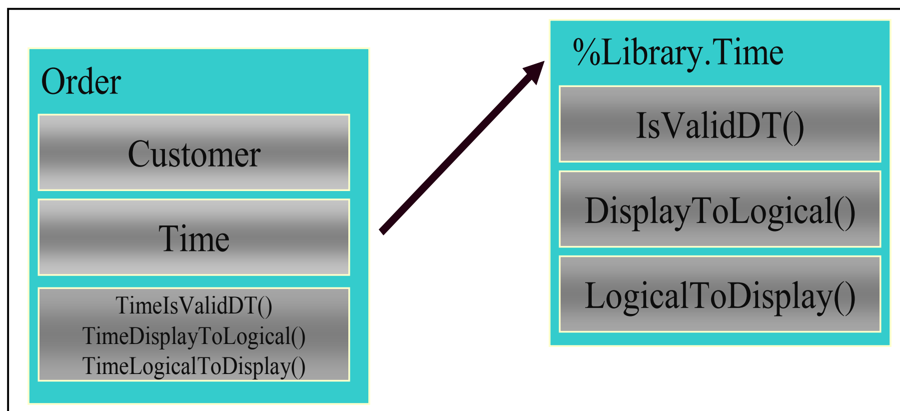
the date will be stored as the user entered it instead of in an internal format, which may cause errors in calculations, or

the date be displayed in a format that will mean nothing to the user.

Unexpectedly, you do not call them like this:

```
do ord.Time.DisplayToLogical() ; this is wrong!
```

The names of methods in datatype classes get combined with the property name that's referring to the datatype, forming a new method that is part of the main class, not the datatype class.



Examples: Datatype Methods

The Order class has Time and Date properties of datatype %Library.Time and %Library.Date, which have their own LogicalToDisplay(), DisplayToLogical(), LogicalToOdbc(), and IsValidDT() methods.

These become the following class methods in the Order class, with the specific data entered as an argument.

```
USER>write ##class(User.Order).TimeDisplayToLogical("1:00 pm")
46800
USER>write ##class(User.Order).TimeIsValidDT(46800)
1
USER>write ##class(User.Order).TimeLogicalToDisplay("46920")
13:02:00
USER>write ##class(User.Order).DateLogicalToOdbc(+$h)
2001-11-16
```

The Time* and Date* methods, like the *Get methods, are not visible anywhere in Object Architect.

Writing Datatype Methods

Create a datatype class by deriving it from a system datatype class (%Library.String, for example).
The class inherits the appropriate datatype methods.
Caché requires standard names for these methods.
Override the methods, delete the inherited code, and write the new methods.
Write an application-specific method for validation.
Write application-specific methods for conversion.
An inherited datatype method is a generator method.
Change its Code Mode Characteristic from Generator to Code.

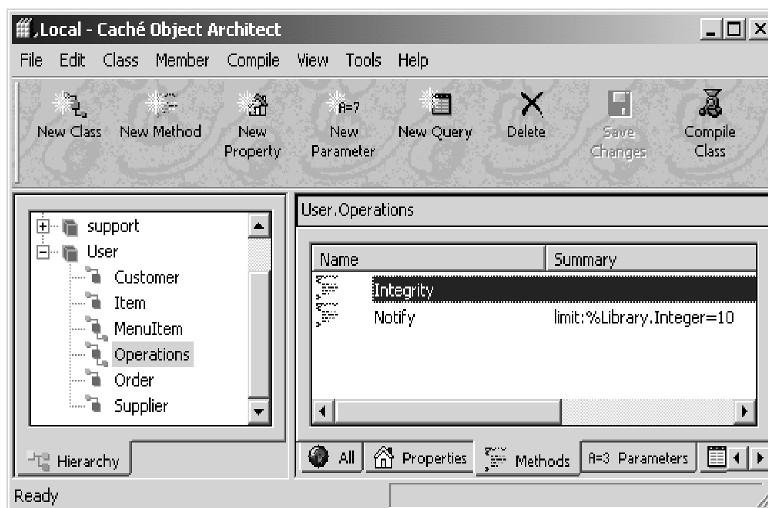
Datatype Population

A user-defined datatype may require a user-defined population method.
Place this method in a separate non-registered class for user-defined population methods.
Do not place these methods in the datatype class.
Call the methods as part of a POPSPEC, specifying the complete <package>.<class> name:

```
CustID:##class(User.PopUtils).IDGen()
```

Other Non-Registered Classes

Use non-registered classes to hold related methods that do the business operations of your company.
The application front-end calls these methods, which isolate the front-end from directly using the persistent classes.
These operations call other classes' methods.



ObjectScript: Random Numbers

The $Random function takes a single argument (*range*), and returns a random integer between 0 and *range*-1.

range is a non-zero positive integer.

Returned numbers are uniformly distributed across the specified range.

EXERCISES: MODULE 9

EXERCISES: MODULE 9

In this exercise, you write some methods for data types of your own: an external ID number that several of your classes share, and methods to help you populate several properties. You create an Operations class for your application to combine the functionality of several persistent classes.

1. Create a new datatype class, IDNum, in the Nothing package, as a class derived from %Library.String. Specify the PATTERN parameter for the class as 3L10N.

2. Change the data type of the CustID, EmpID, StoreID, and OrderID properties to reference the new Nothing.IDNum class.

3. In the IDNum class, override the LogicalToDisplay() method to use $Extract so that it adds "-" characters to turn its argument string that looks like this: XXX########## into this: XXX-#####-#####. Change it from a Generator to a Code method. The method should return the modified string.

```
quit:(%val = "") ""
quit $e(%val,1,3) _ "-" _
     $e(%val,4,8) _ "-" _
     $e(%val,9,13)
```

4. Override the DisplayToLogical() method to use $Translate so that it removes the "-" from its argument. Change it from a Generator to a Code method. The method should return the modified string.

```
quit $tr(%val,"-")
```

5. Create a new non-registered class, PopUtils, in the Nothing package. To create this class, select the Derived option, but don't specify any class name in the text field.

6. In PopUtils, create a Country() class method by copying, pasting, and renaming the %Library.PopulateUtils.City() method. Edit it so that it produces several random country names.

7. Add Country:##class(Nothing.PopUtils).Country() to POPSPEC in the Address class. Notice that since the method is not in PopulateUtils, it needs a package specification.

8. In PopUtils, create a IDGen() class method with one argument (*prefix*, of type %Library.String) that uses $random(99999) + 1, $Justify, and $Translate to create a string that looks like this: XXX##########. The return type should be %Library.String.

```
set num1 = $r(99999) + 1, num2 = $r(99999) + 1
set num1 = $tr( $j(num1,5), " ", "0") ; pad on the left with
                                        zeroes
set num2 = $tr( $j(num2,5), " ", "0") ; pad on the left with
                                        zeroes
quit prefix_num1_num2
```

9. Add CustID:##class(Nothing.PopUtils).IDGen("cus") to POPSPEC in the Customer class, and do the same (using "emp", "sto", and "ord") for POPSPEC in the Employee, Store, and Order classes.

10. In PopUtils, create a CheckGen() class method to generate Check Numbers when populating the Check class. The return type should be %Library.String.

```
quit ($Random(9000) + 1000)
```

11. Right-click on CheckGen() and choose *Copy.* Right-click again anywhere below the list of methods in PopUtils and choose *Paste.* Double-click CheckGen1(). Change its name to CCGen(), and change the code as specified below. Now you have a method to generate CreditCard Numbers when populating the CreditCard class.

```
quit ($Random(9000) + 1000) _ "-" _
     ($Random(9000) + 1000) _ "-" _
     ($Random(9000) + 1000) _ "-" _
     ($Random(9000) + 1000)
```

12. Add the following POPSPECs to the classes below. All the %Library.Date properties will automatically populate correctly. PurchaseOrder.Company will automatically populate with a company name.

Class	**POPSPEC**
Check	Number:##class(Nothing.PopUtils).CheckGen()
CreditCard	Number:##class(Nothing.PopUtils).CCGen()

13. In PopUtils, create a Rebuild() class method with one argument (*count,* of type %Library.Integer, and a default value of 10) for destroying and rebuilding your test database. The return type should be %Library.Status.

```
; delete all the data
   do ##class(Nothing.Customer).%KillExtent()
   do ##class(Nothing.Employee).%KillExtent()
   do ##class(Nothing.Store).%KillExtent()
   do ##class(Nothing.Order).%KillExtent()
   do ##class(Nothing.Payment).%KillExtent()
   ; populate in the proper order
   do ##class(Nothing.Store).Populate(count,1)
   do ##class(Nothing.Customer).Populate(count,1)
   do ##class(Nothing.Employee).Populate(count,1)
   ; generate different kinds of payments
   for i=1:1:count {
        do $case( $r(3), 0:##class(Nothing.Check).Populate(1,1),
                         1:##class(Nothing.CreditCard).Populate(1,1),
                         2:##class(Nothing.PurchaseOrder).Populate(1,1))
         }
   do ##class(Nothing.Order).Populate(count,1)
   do ##class(Nothing.Item).Populate(count*2,1)
   quit 1
```

14. Save and compile all the classes.

15. Using Terminal, test your datatype methods by using the following:

```
      write
##class(Nothing.Customer).CustIDLogicalToDisplay("cus1234567890")
      write ##class(Nothing.Customer).CustIDDisplayToLogical("cus-
12345-67890")
      write ##class(Nothing.Customer).CustIDIsValidDT("cus1234567890")
```

16. Using Terminal, call your Rebuild() method. Use Explorer or SQL Manager to verify that all the new population methods worked.

17. Open any Order. As you did at the end of the last module, enter the following code to test your Payment Process() method (which will also test the population methods again):

```
do ord.Payment.Process(ord.Total)
```

18. In the Customer class, create a new LookUp() class method. The return type should be %Library.Integer. For now it will be a dummy method.

```
read !, "Enter a customer ID: ", custid
quit custid
```

19. In the Customer class, create a new OrdersList() instance method. The return type should be %Library.Integer. This method uses the Customer->Order relationship to show the Customer's Orders if there are any. The method allows the user to choose one of the Orders, and returns the Order ID.

```
set count = ..Orders.Count()
if count = 0 {
    write !, "No orders."
    quit 0
    }
write !, "ID", ?10, "OrderID", ?30, "Total"
for ordcount=1:1:count {
    set ord = cu.Orders.GetAt(ordcount)
    write !, ord.%Id(),
      ?10,
##class(Nothing.Order).OrderIDLogicalToDisplay(ord.OrderID),
      ?30, ord.Total
    }
read !,"Enter an order ID: ", id
quit:id="" 0
quit id
```

20. Create a new non-registered class called Operations, in the Nothing package. In this class, create a Invoice() class method. The return type should be %Library.Status. This method allows the user to pick a Customer (using the LookUp() method). It opens the Customer and allows the user to pick an Order (using the OrdersList() method). It opens the Order and prints a nicely formatted invoice for that Order (for now, to the Terminal).

```
set custid = ##class(Nothing.Customer).LookUp()
quit:custid=0 0 ; no customer
set cus = ##class(Nothing.Customer).%OpenId(custid)
```

```
quit:cus="" 0 ; custid isn't a valid customer ID
set ordid = cus.OrdersList()
; check for no orders
if (ordid = 0) {
    do cus.%Close()
    quit 0
}
set ord = ##class(Nothing.Order).%OpenId(ordid)
; check for a valid order ID
if (ord = "") {
    do cus.%Close()
    quit 0
}
write !!, ord.Customer.Name, !  ;If you have a PrintHome()
                                 method, use it
do ord.Customer.HomeAdd.Print() ;instead of these two lines.
write !!,
##class(Nothing.Order).OrderIDLogicalToDisplay(ord.OrderID),
      ?25, ord.Total, !
write !!
do ord.Payment.Process(ord.Total)
do ord.%Close()
do cus.%Close()
quit 1
```

21. Using Terminal, test the Invoice() class method.

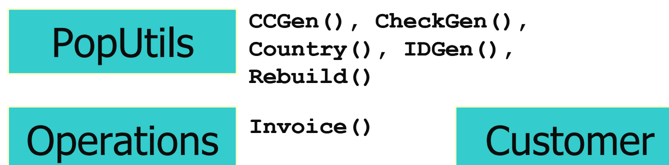
MODULE 10: SQL

10 SQL

Hierarchy: More Classes

Datatype class

Other non-registered classes, and methods they call



Introduction to SQL

SQL began as the language of relational databases (mathematical theory).
It is incorporated into various applications.
Because SQL statements are similar to sentences, the different parts of the statement are called clauses:
the FROM clause,
the WHERE clause, etc.
SQL programs are called stored procedures.
Stored procedures combine SQL with control constructs (If/Then/Else) from the application language (for instance, ObjectScript).
SQL Documentation is available via the Documentation link: *Caché SQL Reference*.

SELECT Queries

SQL can retrieve data via SELECT queries (reporting):

```
SELECT Customer.Name, Order.Time
FROM   Customer, Order
WHERE  ( Order.Status = "Pending" ) AND
         ( Order.Customer = Customer.ID )
ORDER BY Order.Time
```

This query retrieves the names of Customers that are still waiting for their order, and displays them based on how long they've been waiting.
The second part of the WHERE clause is a join.
A join specifies how to join the Customer and Order tables.
Without a join, every Customer would be joined to every order in Order.

The result of a query on one or more tables is also a table.

Aggregate Functions

In the SELECT clause, in addition to simply retrieving data, use aggregate functions, such as:

MIN
MAX
SUM
AVG
COUNT

To use an aggregate function, add a GROUP BY clause to the query.

For example, MIN(Order.Time) using "GROUP BY Customer.Name" won't show actual Order times; it will show only the Customer who's been waiting the longest, and his Order time.

Other SQL Operations

SQL contains commands for updating data or inserting data into a table:

INSERT INTO MenuItem (Name, Price, Quantity)
VALUES ("Chips", 0.75, 30)

SQL also has commands for altering or adding to the structure of the table.

Commands to alter the structure of the table are written in DDL (Data Definition Language).

To issue these commands on a table, check "Allow DDL Operations" on the SQL tab of the class definition.

DDL Examples

Adding columns to the structure of the table:

ALTER TABLE MenuItem ADD COLUMN Cost %Currency

Creating a new table (with 2 columns):

CREATE TABLE Person (Name VARCHAR(35), DOB DATE)

In Caché, these commands do more than update the SQL table definition. They update the class definition by:

adding a property (Cost) to the MenuItem class, and
adding a new class Person with 2 properties.

There are DDL commands to create user accounts and to grant or revoke privileges for each user.

Caché SQL

Caché SQL supports:

Views
Stored procedures
Triggers
Roles/privileges

DDL import
Flat file import

Query Wizard

The Query Wizard assists you in building SELECT queries only.
Hold <Ctrl> and click the New Query button to bypass the Wizard and enter the SELECT query yourself.

Parameters

The Wizard allows you to specify parameters, similar to method arguments.
Parameters are variable elements of the query:

```
WHERE Price > :val
```

At run-time, the user supplies the parameters used during the execution of the query.

Columns

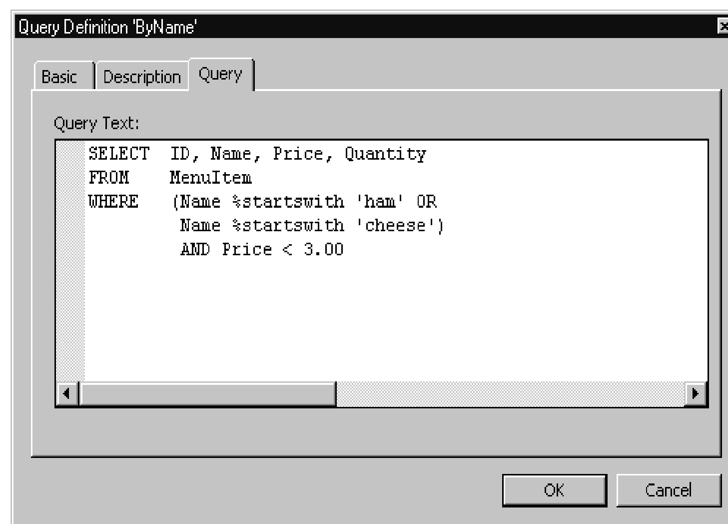
Any properties in the class can be chosen as a column to be included in the query.
Choosing columns builds the SELECT and FROM clauses.

```
SELECT ID, Name, Price, Quantity
FROM   MenuItem
```

The ordering of the properties builds the Row Specification.
The Row Specification is the order in which the columns appear in the table, and the data type of each column.
See the Basic tab of the Query definition.

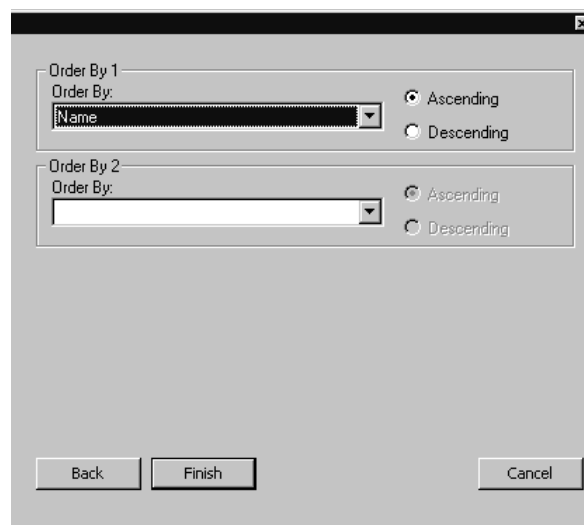
Conditions

Adding conditions builds the WHERE clause.
The Wizard provides:
English versions of standard SQL logical operators ("greater than" instead of ">").
The Caché %startswith operator (similar to the SQL "LIKE" operator).
WHERE Name %startswith 'SM,J'
The Wizard doesn't give access to the Caché %pattern operator.
To use a pattern, add it to the query directly.
WHERE Zip %pattern '5n.1(1"-"4n)'
Complex Conditions
Complex conditions using AND and OR are possible.
If you need parentheses to clarify or control the evaluation, enter them yourself.



Order By

The Order By fields sort the result set by the fields selected.



Query Details

Double click on a Query name in Object Architect to bring up the details of the query.
Information can be viewed or modified from this form.

Row Specification

The Row Specification describes the format of the records retrieved by the query.
The ID Column Index is the column position of ID in the query, or 0 if the query doesn't return ID.
You must keep the Row Specification, the ID Column Index, and the SELECT clause in sync.
See:
file:///C|/CACHESYS/Docs/dev/devclassspec.html#24958 for more about Row Specification.
Note: This URL works only if C:\CacheSys is your installation directory.

Views

A SQL View is a virtual table.
To build a view:
Create a query and select the SQL View option on the Basic tab, or
Use an SQL Create View statement.
A view built by creating a query can not have parameters.
A view is a subset of data in the table(s) being queried, always up to date. It may itself be queried.
Unlike regular tables, a view can't be used for data entry.
Use it only to retrieve data.

Stored Procedures

A method or a query may be projected via the SQL projection as an SQL stored procedure.
Click the option in the method or query definition.

Using SQL

Use SQL in a number of ways within Caché.
Use the Query Wizard, or your own SQL knowledge, to build a SELECT query using SQL.
Use the %Library.ResultSet class (covered next slide) to access your query.
Use Embedded SQL to write an ObjectScript program that contains SQL.
Assemble a query at run-time (called a Dynamic Query) and use it with the %Library.ResultSet class.
Send SQL into Caché via ODBC/JDBC.

ResultSet Class

Caché provides the %Library.ResultSet class for executing queries and processing the results.
Pass parameters to the query as arguments of the Execute() method.
For example:

```
set rs = ##class(%Library.ResultSet).%New("User.MenuItem:
ByName")
do rs.Execute()
while (rs.Next() '= 0) {
      write !, rs.Get("Name")
      write ?40, rs.Get("Price") }
do rs.Close()     ; close this query
do rs.%Close() ; close the ResultSet object
```

Runtime Mode

Use the RuntimeMode property of a ResultSet object to determine the format for the data retrieved. Values are:
0 (Logical)
1 (ODBC)
2 (Display)
The Get() method retrieves each column's data according to the RuntimeMode.
USER>write rs.Get("DOB") ; using RuntimeMode = 2
Dec 23, 1949

Embedded SQL

Create any non-SELECT query by embedding valid SQL code in an ObjectScript program.
Or create SELECT queries using embedded SQL within ObjectScript. Use a cursor to access each record that the query retrieves.
The process is similar to using the %Library.ResultSet class.
Reference ObjectScript variables inside the SQL by placing a colon in front of the variable.

Embedded SQL SELECT Queries

First, declare the cursor:
```
&sql(DECLARE ExpFood CURSOR FOR SELECT Name, Price FROM MenuItem
WHERE Price > :limit)
```
Next, open the cursor ("run" the query):
```
&sql(OPEN ExpFood)
```
Next, within a loop, fetch data into variables, and use them:
```
for  {
      &sql(FETCH ExpFood INTO :food,:price)
      quit:(SQLCODE '= 0)
      write !, food, ?40, price }
```
Finally, close the cursor:
```
&sql(CLOSE ExpFood)
```

SQLCODE and %ROWCOUNT

Check the variable SQLCODE after each FETCH to determine when to exit the loop. Its values are:
0: a record was found (keep looping)
100: no more records
<0: failure, with the specific error message documented in the Documentation link: Caché Error Messages Reference.
After each FETCH , the %ROWCOUNT variable contains the current count of records returned by the query.

Dynamic Queries

Queries built using the Wizard or using Embedded SQL are compiled at the time the class is compiled.
A Dynamic Query can be assembled at run-time.
SELECT queries as well as any other SQL statements are supported.
If the query creates a table, the %msql variable must be set to a defined SQL user, such as "_system".
Use %Library.ResultSet and its methods as with a Wizard-built query. There are several differences:
Use of the %Library.DynamicQuery:SQL query.
Use of the Prepare() method.

Query parameters.

Using Dynamic Queries

Start by creating the result set object:

```
set rs = ##class(%Library.ResultSet).%New("%Library.
DynamicQuery:SQL")
```

Next, build the query string:

```
set sql = "SELECT Name, Price FROM MenuItem WHERE Price > ? AND
Name %startswith '?'"
```

Each “?” in the query string becomes a parameter. The first “?” is the first parameter of the Execute() method, the second “?” is the second parameter, etc.

Using “?” for parameters, rather than substituting exact values when building the query string, results in more efficient query code.

```
Next, prepare the query:
set st = rs.Prepare(sql)
Do rs.Execute(50, "S") ; price>50,name starts with S
```

Triggers

The SQL tab of a class definition allows you to specify SQL triggers. These are methods that Caché calls when certain SQL events happen.

They are **not** called from objects (see Callback Methods in Module 8).

There are 6 triggers:

BEFORE INSERT
AFTER INSERT
BEFORE UPDATE
AFTER UPDATE
BEFORE DELETE
AFTER DELETE

ObjectScript: While Loops

A While loop terminates based on a condition checked before the code executes.

while <condition> {code}

Enclose code in curly braces {}.

A quit within the code also terminates the loop.

A Do/While loop terminates based on a condition checked after the code executes.

do {code} while <condition>

EXERCISES: MODULE 10

EXERCISES: MODULE 10

In this exercise, you use the Query Wizard and Embedded SQL to create several SQL queries that provide needed functionality to your application. You also create a Dynamic Query that adds a new table (and class) to your application.

1. Add a FindByName query to the Customer class that takes a parameter (*name,* of type %Library.String), includes the ID and Name columns, orders the results by Name, and has the condition that the Name starts with *name*.

2. Edit the LookUp() method in the Customer class to prompt for a partial name. Then, use a ResultSet object and the FindByName query to display the matches. The user can then pick the ID of one of the matches. The code below shows 12 lines to be added before the 2 that are already in the method.

```
read !, "Enter a partial name: ", nm
      write "...searching..."
      set found = 0 ; assume no matches
      set rs =
##class(%Library.ResultSet).%New("Nothing.Customer:FindByName")
      do rs.Execute(nm)
      while (rs.Next() '= 0) {
      set:(found=0) found = 1 ; at least one match!
          write !, rs.Get("Name")
          write ?25, rs.Get("ID") }
      do rs.Close()     ; close this query
      do rs.%Close()    ; close the ResultSet object
      quit:(found=0) 0 ; no matches
      read !,"Enter a customer ID: ", custid
      quit custid
```

3. Save and compile the Customer class. Using Terminal, test your new query and method by running Invoice().

4. Add a ByOrder query to the Item class that takes a parameter (*order*, of type %Library.Integer), includes the ID, Quantity, Size, and Price fields of an item, and has the condition that Order equals the *order* parameter. **Note**: After the Query Wizard creates the query, you will have to edit the query and place quotes around the Order and Size columns ("Order" and "Size").

5. In the Order class, write a ItemsList() **instance** method that uses a ResultSet object to print the data from the ByOrder query of the Item class (passing in the ID of the Order). The return type should be %Library.Status.

```
      set rs =
##class(%Library.ResultSet).%New("Nothing.Item:ByOrder")
      do rs.Execute(..%Id())
      while (rs.Next() '= 0) {
           ; display the ID, Quantity, Size, and Price, using the
             Get() method
            }
      do rs.Close()     ; close this query
```

```
do rs.%Close()    ; close the ResultSet object
quit 1
```

6. In the Operations class, add a line to the Invoice() method, just after writing the OrderID and Total, to call ItemsList(), on the open Order, to display the Items of the Order. Run Invoice() again to test this.

```
do ord.ItemsList()
```

7. Write a Pets() **instance** method in the Customer class, that uses Embedded SQL to print a list of the Pets of a Customer. The return type should be %Library.Status.

```
set cust = ..%Id() ; get the ID of the customer
&sql(DECLARE PetCur CURSOR FOR
     SELECT Customer->Name, Pets, element_key
     FROM Nothing.Customer_Pets
     WHERE (Customer = :cust))
&sql(OPEN PetCur)
; Note: place 2 spaces between the for command and the {
for  {
    &sql(FETCH PetCur INTO :owner, :pet, :dob)
    quit:SQLCODE'=0
    ; show the owner, pet name, and the pet (external) DOB in
      columns
    }
&sql(CLOSE PetCur)
quit
```

8. Save and compile the Customer class.

9. Test your Pets() method, by opening a Customer and calling the method.

10. Using Studio, create a routine called *promo* that uses a ResultSet object to execute the following SQL statement as a Dynamic Query.

```
promo ; create Promotion class
set %msql = "_SYSTEM" ; required! Set the OWNER of the table
set sql = "CREATE TABLE Nothing.Promotion (" _
      "Discount NUMERIC (2,2)," _
            "EndDate  DATE," _
      "Slogan   VARCHAR (200)," _
            "StartDate DATE," _
            "Stores   Nothing.Store )"
set rs =
##class(%Library.ResultSet).%New("%Library.DynamicQuery:SQL")
do rs.Prepare(sql)
do rs.Execute()
do rs.Close
```

As an added challenge, consider adding some status checks for the Prepare and Execute routines.

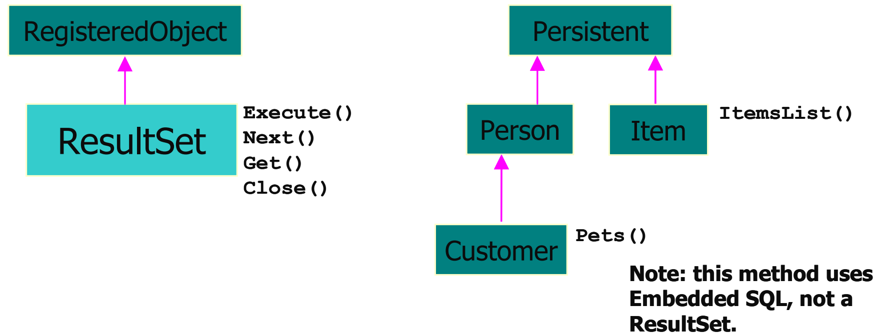
11. Save the routine as promo.MAC. Compile it (Build->Compile). Run it (do ^promo) using the Terminal.

12. This ResultSet does not retrieve data; it creates a table. Look at the table in SQL Manager. Then re-connect Object Architect and view the new class. Edit the Stores property, make it a Relationship (Many-to-One), and save and compile the Promotion class.

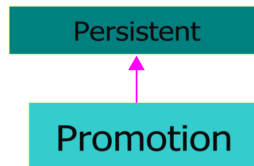
MODULE 11: INDEXES

11 INDEXES

Hierarchy: %ResultSet Class



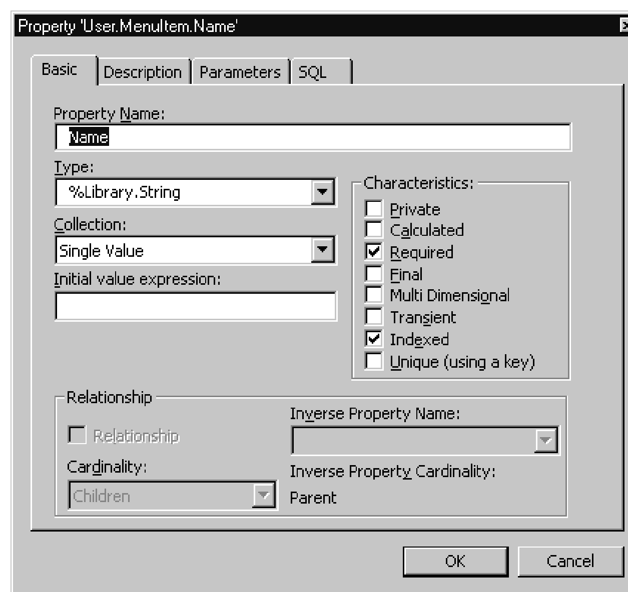
Hierarchy: Promotion Class



Indexes

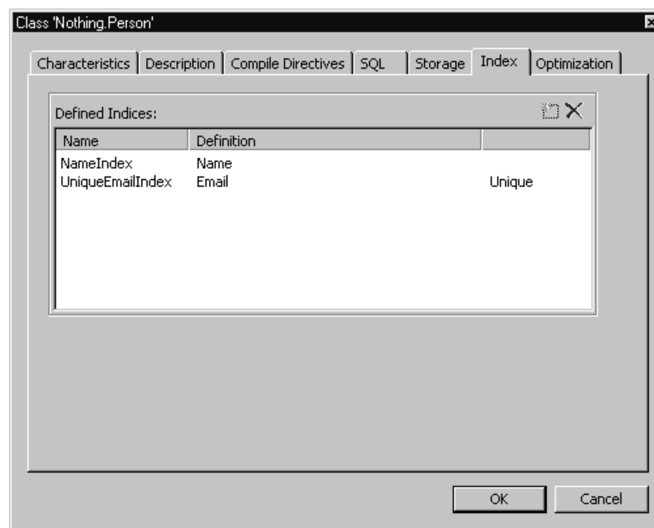
You've already seen how easy it is to add indexes to a class: simply click the "Indexed" option for a property that you want to index.

To further customize the behavior of an index, or for more complex indexes, double-click the class and click on the Index tab.



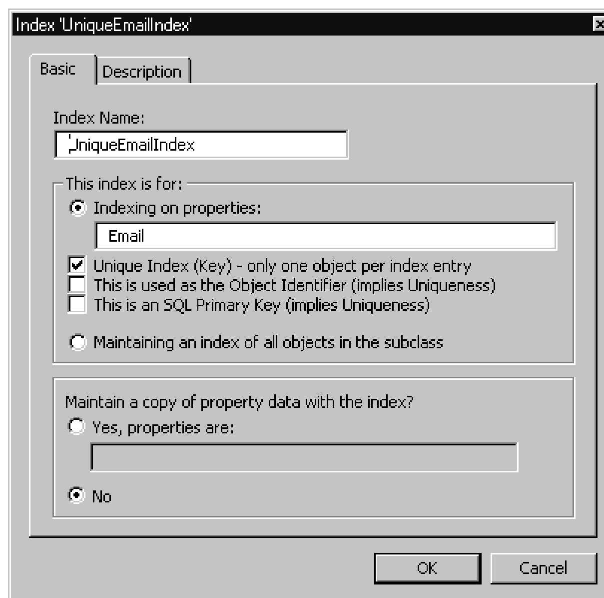
Index Tab

Clicking the "Indexed" or "Unique" options for a property adds an entry on the Index tab.
You can customize an index by double-clicking it.
You can add one or more complex indexes from here as well.



Customizing an Index

This is an example of an "Email" index.
You can list several properties for indexing instead of just one.
You can store other properties' data with the index.



Collation

Each indexed property must have a collation type, which affects how the data will be sorted in the index.

This is assigned via the property's COLLATION parameter:

ALPHAUP - Removes all punctuation characters except question marks ("?") and commas (",") then translates all the lowercase letters to uppercase.

EXACT - No translation is performed.

MINUS - Makes the value numeric and changes its sign.

PLUS - Makes the value numeric.

SPACE - Translates the value into a string.

STRING - Converts a logical value to upper case, strips all punctuation and white space (except for commas)

UPPER - Translates all lower case letters into upper case letters.

Caché Uses Indexes

A query with a Where condition on an un-indexed column (Order.Status) searches all records to retrieve its results:

```
SELECT  Name, Phone
FROM    Customer, Order
WHERE   ( Order.Status = "Pending" ) AND
        ( Order.Customer = Customer.ID )
```

Select the Indexed option for the property.

Use SQL Manager to rebuild indexes for any class with a newly indexed property, if the class already contains data.

Caché uses the index to locate the matching records.

Better Generated Code

Caché generates faster code when it can use an index.

Section of some code for a query without an index, using `^Nothing.OrderD`:

```
set %objcsd(qHandle, 6) = $order(
^Nothing.OrderD(%objcsd(qHandle, 6)))
if %objcsd(qHandle, 6) = "" goto %0ABdun
set %objcsd(qHandle, 12) = $get(
^Nothing.OrderD(%objcsd(qHandle, 6)))
```

Section of some code for the same query with an index, now using `^Nothing.OrderI`:

```
set:(%objcsd(qHandle,3) = "") %objcsd(qHandle, 3) = -1E14
set %objcsd(qHandle, 6) = $order(^Nothing.OrderI("Status",
      %objcsd(qHandle, 3), %objcsd(qHandle, 6)))
if %objcsd(qHandle, 6) = "" goto %0ABdun
```

EXERCISES: MODULE 11

EXERCISES: MODULE 11

In this exercise, you create some more needed queries, and investigate the effect of indexes on the queries.

1. In the Order class, add a Statement query that has one parameter (*id*), selects Customer.Name and Customer.Email, ID, OrderID, Date, and Total, has the condition that the Customer equals the *id* parameter, and sorts by Date. Edit the query so that the FROM clause is Nothing."Order", and place quotes around "Date" in both places. (Hint: select ID, OrderID, Date and Total before opening Customer to get Name and Email. Then rearrange the order of selections.)

2. Save and compile the Order class.

3. Using SQL Manager, use Object->Execute Query to test your Statement query on a specific Customer. Copy-and-paste it from the class definition into the Execute Query text box. For testing purposes, change the ":id" in the Where clause to a specific ID for a Customer. Click the Execute button and verify that the query works.

4. Back in Object Architect, use the Index tab in the Order class to add an Index called Billing. It should index Customer.

5. Save and compile the Order class.

6. Right-click on the Order table in SQL Manager and rebuild the indexes for this table. Use Explorer to look at the ^Nothing.OrderI global.

7. Add an Index called Rewards to the Order class. It should index Employee.

8. In the Order class, add a Performance query that has one parameter (*id*), selects Employee.Name and Employee.Email, ID, OrderID, Date, and Total, has the condition that the Employee equals the *id* parameter, and sorts by Date. Edit the query so that the FROM clause is Nothing."Order", and place quotes around "Date" in both places.

9. Save and compile the Order class.

10. Use SQL Manager to test your Performance query. Why doesn't it find any data? What should you do about it?

MODULE 12: CACHÉ SERVER PAGES

12 CACHÉ SERVER PAGES

Caché Server Pages

Caché Server Pages (CSP) allows a developer to build a web application that:
extends the functionality of HTML pages with additional tags and attributes,
has event-driven links between the local browser and the remote Caché server, and
calls both Javascript and ObjectScript code.
Developers can further customize the functionality of their application by creating their own tags.

Programming Elements

Programming elements of a CSP Page include:
CSP tags (pseudo-HTML),
CSP attributes (pseudo-HTML) used within standard HTML tags,
Custom CSP tags, and
Caché ObjectScript code, using either simple expressions or complex procedures.

How does it work?

When a CSP page is used:
A browser requests a Caché Server Page.
The Web Server passes the request to the correct Caché Server for processing.
Caché runs the generated routines associated with the page, and produces an HTML page.
The HTML page is sent via the Web Server back to the browser.
The Browser, the Web Server and the Caché Server can be on one machine, or on several machines in any combination.

Benefits

CSP operates faster than other Web applications because each component does what it does best.

The browser makes requests and displays results.

The Web Server serves standard HTML pages.

The Caché Server handles the processing and database access with its usual caché.

This system is efficient because the code accesses the data natively.

There is less passing of data back and forth.

How do you make it work?

Look at the configuration on your system right now.

Your machine is acting as browser, web server, and Caché server simultaneously.

The URL is: http://localhost/csp/<your namespace>/person.csp.

Localhost is an alias that refers to your machine as a web server.

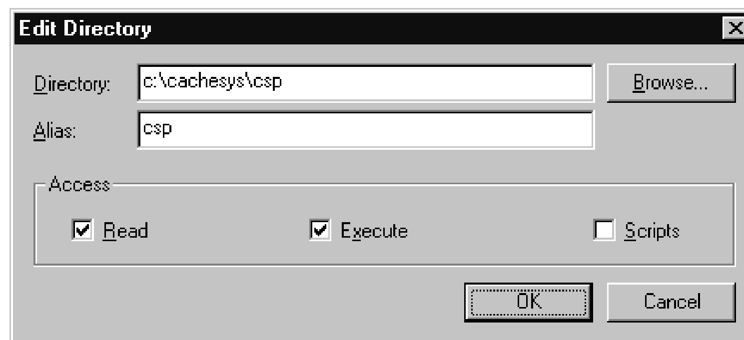
The first /csp is a defined Virtual Directory that triggers the web server.

The final .csp tells the web server to pass the job on to CSP Gateway.

Virtual Directory

Cache installation creates a web server virtual directory "/csp" which represents "c:\cachesys\csp", with Read and Execute privileges.

For CSP, this directory simply allows the web server to respond to requests that start with //localhost/csp.



Caché Takes Over

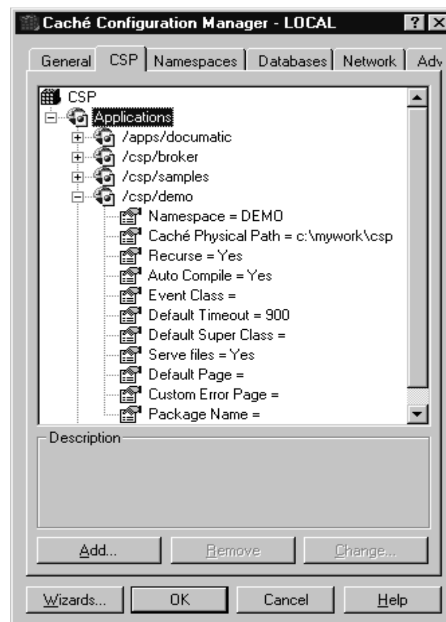
When the web server sees that the request ends in ".csp" it passes the request to the CSP Gateway.
The Gateway, and the web server's ability to call it, are set up by Caché installation.
The Gateway's job is to pass the request to the correct Caché server (since the web server could be referencing more than one Caché server).
The Gateway needs to know only 2 things:
which Caché server this application uses, and
the IP address of the Caché server.

CSP Gateway

The CSP Gateway (CSPmssys.dll) allows management of CSP on a web server.
Caché installation configures the CSP Gateway in two ways:
An application "/csp" that uses a server called "LOCAL".
A server called "LOCAL" with IP address "127.0.0.1".
Since the request contains "/csp", CSP knows to pass the request to the Caché server on your machine.

Caché Configuration

At Namespace creation, the Configuration Manager adds a CSP Application, called /csp/<your namespace> ("/csp/demo" in the example at right).
The URL points Caché to your namespace.
Initially, there are no generated routines in the namespace.
Caché uses the Physical Path, to find the person.csp source in c:\mywork\csp.



Code Generation

Caché reads the person.csp file and creates a class called csp.person (the person class in the csp package), containing methods to render HTML.

```
(class) Nothing.Person ->(page) person.csp ->(class) csp.person
```

When Caché compiles the class, it generates the code for the methods.
Caché runs the generated code.
Subsequent requests for the same page will run the same code.
Caché will re-generate the code only when the CSP source page changes.

Form Wizard Source Code

The Form Wizard generates a CSP page including:
HTML
CSP tags
JavaScript functions
When compiled, the page generates a class (csp.<class>) on the Caché server, complete with methods.
Some of these methods:
run when the page is first called and generate HTML and JavaScript, which is sent via the web server to the browser.
are called from the page when it is already running in the browser.

Dreamweaver Integration

As Dreamweaver starts, it displays a “Connect to Caché” dialog box.
Dreamweaver uses the selection to make a live connection to the selected Namespace in Caché.

The "Caché CSP" item added to the Insert menu allows the developer to place CSP tags on a CSP page.

CSP Features

There are many features of CSP that are beyond the scope of this class. Those we will discuss are:

The CSP Form Wizard
Preformatted Dynamic Data Display
Searches and Queries
HyperEvents (Event handling)

The CSP Form Wizard

The simplest way to create a complete CSP page is by using the CSP Form Wizard.

From Dreamweaver: Insert>Caché CSP->Form Wizard.
Directly: Run CacheWebFormWizard.exe. (in c:\cachesys\bin)

Specify a Caché connection, then a class within that namespace.

Pick the properties for display on the page.

The Wizard creates a CSP page (source code) that contains text boxes for the properties, along with New, Save, and Search buttons. In Dreamweaver:

use the menu to insert additional objects, like buttons.
select View->Code to modify the HTML code directly.

The <csp:Object> Tag

The Form Wizard uses a <csp:Object> tag to instantiate an object in a CSP page. That is, when the page is requested, the object's data is available to the page.

For example, this instantiates object #4 of the MenuItem class, assigning "mi" as the oref:

```
<csp:object name="mi" classname="User.MenuItem" objid="4">
```

This is the CSP equivalent of

```
set mi = ##class(User.MenuItem).%OpenId(4)
```

If objid has no value, a new object is instantiated.

Naming CSP Objects

The Form Wizard assigns "objForm" as the name of the <csp:Object> tag, as well as the cspbind attribute of the <form> tag.

These two attributes must match. This binds the input fields of the form to the properties of the object.

The <csp:Object> name attribute ("objForm" or another more specific name) is an

```
oref. Use it as you would any oref.
objForm.Price
objForm.CheckStatus()
```

The %request Object

The %request object allows one page to pass data to another. The <csp:Object> tag uses the following attribute:

objid=#(%request.Get("OBJID"))#

The expression %request.Get("OBJID") initially has no value.

The Search page sends the id of the chosen instance to the original page, in the OBJID variable, via the %request object.

The original page then instantiates the object selected on the Search page, and displays its data.

Dynamic Data

CSP expressions are the simplest way to display dynamic content on a page.

When a CSP page is requested, the current value of the expression appears on the page.

The expression syntax #( )# works for any kind of ObjectScript expression.

This expression displays the current date:

```
#($zd($horolog,2))#
```

This expression displays a property of an object that has been opened:

```
#(mi.Price)#
```

Expressions within CSP tags also supply dynamic values for attributes.

A Search From the Browser

A search is initiated by a button and controlled by a `<csp:Search>` tag.

The onClick attribute of the button calls the JavaScript function named by the `<csp:Search>` tag.

The `<csp:Search>`, when accessed by an onClick event, creates a new page at run-time.

```
<input type=button name=btnSearch value=Search
onClick="form_search();">
<csp:search name="form_search" classname="User.MenuItem"
where="Name,Price">
```

The page retrieves objects from the database, allowing the user to retrieve information.

The where attribute specifies a list of properties to be displayed.

An SQL Query

A pre-defined SQL query is run from a `<csp:Query>` tag.

The `<csp:Query>` tag names the query to be run, and lists any supplied parameters.

When adding `<csp:Query>` to a page, check the “Create Table” option in Dreamweaver.

The `<csp:While>` tag will be used several times to create a table for the results.

The tag below runs the ByName query of the MenuItem class, displaying items that start with "s":

```
<csp:query name="query" classname="User.MenuItem" queryname="ByName" p1="s">
```

Note that the p* parameters correspond to the parameters that are defined in the ByName query.

A pre-defined SQL query is run from a `<csp:Query>` tag.

The `<csp:Query>` tag names the query to be run, and lists any supplied parameters.

When adding `<csp:Query>` to a page, check the "Create Table" option in Dreamweaver.

The `<csp:While>` tag will be used several times to create a table for the results.

The tag below runs the ByName query of the MenuItem class, displaying items that start with "s":

```
<csp:query name="query" classname="User.MenuItem" queryname="ByName" p1="s">
```

Note that the p* parameters correspond to the parameters that are defined in the ByName query.

A ResultSet for a Query

The name attribute of the <csp:Query> tag is instantiated as a %Library.ResultSet object.

Use any of the %Library.ResultSet methods to customize the query:

- Next() moves from one member of the result set to the next.
- Get("Name") retrieves the Name of each MenuItem.
- GetColumnCount(), GetColumnHeader() and GetColumnName() are used to display the table of results.

HyperEvents

A HyperEvent calls a method on the server triggered by an event in the browser.

A button's onClick event calls a method:

```
<input type="button" name="btnAdd" value="Add 50"
onClick="#server(Nothing.Customer.Populate(50))#">
```

Use the #server( )# syntax in an event handler to call a method on the Caché Server:

The event handler uses:

```
#server(<package>.<class>.<method>)#
syntax instead of: #server(##class(<package>.<class>).<method>)#
```

Error Handling

CSP errors come from several sources.

Caché Server errors

Caché Server Configuration errors

CSP Gateway Configuration errors

Errors on a CSP page trigger:

the display of the CSP Error page, containing information about the error, or

The display of the requested page, with an embedded error message.
Communication errors between the browser and the web server are handled in the standard way.
The browser displays default error page, or
customize your CSP application so that the user sees your own error pages (custom messages, format).

Useful URLs

http://localhost/csp/bin/cspmssys.dll
Use to access CSP Gateway system management functions.
http://localhost/csp/samples/menu.csp
Use to see sample CSP pages and ways to use different CSP tags.
http://localhost/csp/samples/rulemgr.csp
Use to learn about standard CSP tags and their syntax.

Topics for CSP Course

More CSP tags
Maintaining State
Scripting
Security
Custom tags (Rules)
More Error Handling
Configuration

EXERCISES: MODULE 12

EXERCISES: MODULE 12

In this exercise, you create a CSP page that allows you to edit Customers, and make some modifications.

1. Start Dreamweaver. Connect to your namespace. Use the CSP Form Wizard to create a page based on the Customer class, using all the properties except: Pets, LastName, HomeAdd, WorkAdd, Notes, %Id(), HomeAddress.%Id() and WorkAddress.%Id(). Note: in order to choose the properties of the Home and Work Addresses, you must expand HomeAdd and WorkAdd and select the individual properties.

2. Place the properties in the order in which they should appear on your web page. For instance, place Name first and place Street before City and State. Save your page as *cust.csp* in *d:\mywork\csp*.

3. Click the 2nd icon near the top of the page, which represents the <csp:Object> tag. An edit box reflects the selected object. An icon on the right looks like a pencil stub. Click it to see the entire <csp:Object> tag. Modify the name attribute from *objform* to *cust*. This variable will be used later as an oref.

4. From the Dreamweaver menu, switch from View->Design to View->Code. Find the <form> tag under the <Search> tag. Modify the cspbind attribute of the <form> tag from *objform* to *cust*.

5. Return to the Design. Click the 4th icon near the top of the page to show the edit box for the <csp:Search> tag. Modify the where attribute to allow searches by CustID in addition to Name and Email.

6. Place your cursor at the very bottom of the page, and press <Enter> several times to make some room. Then, use Insert->Cache CSP->Query to insert a <csp:Query> tag that references the Statement query of your Order class. Accept the default ResultSet name of *query*, specify the P1 argument as #(cust.%Id())#. Remember that *cust* is the name of the oref for the page. Select the *Create Table* option and click OK.

7. Use a cursor to show placement and the Insert->Form Object->Button to add a button below the initial three buttons (this creates an <input> tag). Double-click it to edit its properties as specified below

Attibute	**Value**
Name	btnAdd
Label	Add 50
Action	None

8. Then edit the <input> tag by right-clicking on the button: add an onClick event which calls the Populate() class method, passing 50. Use the #server()# syntax:

```
onClick="#server(Nothing.Customer.Populate(50))#"
```

9. Use Insert->Form Object->Button to add another button, next to the Add 50 button (this creates another <input> tag). Double-click it to edit its properties as specified below. Then edit the <input> tag: add an onClick event which calls the %KillExtent() class method. Use the #server()# syntax again.

Attibute	**Value**
Name	btnDel
Label	Delete All
Action	None

10. Save your page.

11. Start Internet Explorer and point your browser at http://localhost/csp/<your namespace>/cust.csp. Add, lookup, and edit entries. Check all fields with good and bad input. Note the Statement data displayed in a table. Try the Search button, as well as your two new buttons.

EXERCISES: MODULE 12
WITHOUT DREAMWEAVER

EXERCISES: MODULE 12 WITHOUT DREAMWEAVER

In this exercise, you create a CSP page that allows you to edit Customers, and make some modifications.

1. The CacheWebFormWizard.exe is found in c:\cachesys\bin. Use it to create a page based on the Customer class. Use all the properties except: Pets, LastName, HomeAdd, WorkAdd, Notes, and %Id(), Home.Address.%Id() and WorkAddress.%Id(). Note: in order to choose the properties of the Home and Work Addresses, you must expand HomeAdd and WorkAdd and select the individual properties. Also choose the order of the properties. For instance, put Name first and put Street addresses before City and State. They will appear in the same order on your web page. Save your page as *cust.csp* in *d:\mywork\csp*.

2. Open Internet Explorer and point your browser at http://localhost/csp/<your namespace>/cust.csp. If you cannot access the page from there, try replacing localhost with localhost:1972.

3. Once your have the web page, do a search on names starting with s. If your database is full, you will probably have at least one Customer beginning with s. Double-click on one Customer and edit his name. Save.

4. Find the code built by the Wizard. It is still just where you saved it. Don't double-click on it. Right-click and open it with Notepad. Find the <csp:object> tag which has in it name = "objForm". Change *objForm* to *cust*, for a more familiar oref.

5. Scroll down a bit more and find the <form> tag. In it cspbind = "objForm". Again, change *objForm* to *cust*.

6. Scroll down to the very bottom. Alright, not quite the bottom. Look for input type = button. There will be 3 of them. Copy one and add two more versions of it immediately below. Be sure all the input tags are before the </TD> tag. For the first new button, change name to btnAdd and value to Add 50. For the onClick, use the #server()# notation. Make sure that all parentheses, brackets and #'s which are opened are also closed. The method will be Populate(50).

```
onClick="#server(Nothing.Customer.Populate(50))#"
```

7. For the second button you have added, make the name btnDel, the value Delete All and the method %KillExtent.

8. Save your page.

9. Use Internet Explorer to go to the same page again. It should have changed. Add, lookup, and edit entries, all from your new page. Check all fields with good and bad input (i.e., no name, future birth date). Try the Search button, as well as the two new buttons. Kill all data and do a search, using the startswith option. Then create new data and repeat the search.

10. You can add a query without an editor, but it's more complicated than a simple method call. In addition to calling the query, you must build a table to exhibit the results.

11. Begin with the query tag. It should look like this, with the parameter using the oref you assigned above:

```
<csp:QUERY NAME="query" CLASSNAME="Nothing.Order"
QUERYNAME="Statement" P1=#(cust.%Id())#>
```

12. Give it an opening table tag that looks like this:

```
<table border=1 bgcolor="">
```

13. If you are fluent in HTML, give it a more interesting background color.

14. Next, it needs an opening <tr> (table row) tag. Then nested <csp: while> tags to build the table across and down. Each while needs a counter and the counter is used to set the condition for the while (tell it when to stop looping). The results will look like this:

```
<tr><csp:WHILE COUNTER=queryCol
            CONDITION="(queryCol<query.GetColumnCount())">
<th align=left><b>
#(query.GetColumnHeader(queryCol))#
</b></th>
</csp:WHILE> </tr>
<csp:WHILE COUNTER=queryRow CONDITION=query.Next()>
<tr class=#($S(queryRow#2:"DarkRow",1:"LightRow"))#>
csp:WHILE COUNTER=queryCol
            CONDITION="(queryCol<query.GetColumnCount())">
<td>
#(query.GetData(queryCol))#
</td>
</csp:WHILE> </tr>
</csp:WHILE>
</table>
```

The and later bold the column headings. The <th> tags are for table headers and the <td> for table data. Notice that the <while> for table header ends </while> before the table row <tr> begins. Be sure that all your tags <tag> have closing tags </tag>s in the right places, nesting as appropriate.

15. Save your csp page, then go back to the IE and call it up again. Find a Customer in your SQL manager who definitely has Orders. Begin by locating his ID in the Order Table. Then find his name in the Customer Table. Do a go back to the IE page, search for your chosen Customer and double-click on his name. When you are back on the original page, all his data should be there, with his Orders listed at the bottom.

MODULE 13: VISUAL BASIC

13 VISUAL BASIC

Visual Basic

Visual Basic (VB) is an object-oriented, event-driven system.
A VB application consists of one or more forms.
The form contains controls: text boxes, buttons, check boxes, lists. Even the form itself is a control.
Controls are objects:
They have properties: the Caption of a label, the width of a text box.
Controls also have methods.
Controls respond to events: typing in a text box, clicking a button.
Most of a VB application's code is in event subroutines.

Factory Object

From VB, a Caché Factory object is used to create and manage Caché objects.
An application need only create one instance of the Caché Factory class.
First, declare a Factory object.

```
Dim objFact as CacheObject.Factory
```

Next, create a new factory and use it to connect to Caché. Insert this code into Form_Load method.

```
Set objFact = New CacheObject.Factory
con = objFact.Connect("cn_iptcp:127.0.0.1[1972]:USER")
If Not con Then
MsgBox "Cannot connect to Cache."
End
End If
```

When the application quits, Form_Unload, disconnect from Caché.
objFact.Disconnect

Visual Basic Syntax

Most ObjectScript syntax is valid in VB.
Certain methods have a different syntax in VB.
Any method that starts with % in ObjectScript must start with sys_ in VB.
Calling class methods is a two-step process in VB.
The table on the next slide summarizes the differences.

Syntax Table

ObjectScript	Visual Basic
set mi.Price = 2.50 write mi.Price	mi.Price = 2.50 txtPrice.Text = mi.Price
set mi = ##class(MenuItem).%New()	set mi = objFact.New("MenuItem")
set mi = ##class(MenuItem).%OpenId(3)	set mi = objFact.OpenId("MenuItem", 3)
set st = mi.%Save()	mi.sys_Save
do mi.%Close()	mi.sys_Close
do ##class(MenuItem).%KillExtent()	set st = objFact.Static("MenuItem") st.sys_KillExtent
do ##class(MenuItem).Populate(50)	set st = objFact.Static("MenuItem") st.Populate 50

Opening/Saving an Object

Once you've opened an object and displayed its data on a form, there are four copies of the object:

the object's data on local disk,
the object's data on the form,
the object's data in memory on the Caché server, and
the object's data in memory on the Visual Basic client.

The objects on the Caché server and the VB client are linked. Changes made to the object on the VB client are immediately sent to the server.

When the user clicks the Save button, the data on the form is copied into the VB object, which automatically copies it into the object on the server.

`sys_Save` (%Save) then saves the data to disk.

Custom Controls

Caché adds two custom controls to the Visual Basic environment, that you can associate with a query:

the CacheList control, which is a grid, and
the CacheQuery control, which creates a form at run-time containing several controls (including the CacheList control).

The Form Wizard automatically uses the CacheQuery control, via the Find button.

Result Sets in Visual Basic

There are three ways to use the result set of a query in VB. Note the differences in syntax.

Use the Factory to create a ResultSet object.

```
Set rs = objFact.ResultSet("User.MenuItem", "ByName")
rs.Execute
```

Use the CacheList control, which has a Factory property, and ResultSet and Run methods.

```
CacheList.Factory = objFact
CacheList.ResultSet "User.MenuItem", "ByName"
CacheList.Run
```

You can use the CacheQuery control, which has Factory, ClassName, and QueryName properties, and a FindId method:

```
CacheQuery.Factory = objFact
CacheQuery.ClassName = "User.MenuItem"
CacheQuery.QueryName = "ByName"
id = CacheQuery.FindId
```

Dynamic Queries in Visual Basic

There are two ways to use the result set of a Dynamic Query in VB. Note the differences in syntax.

Use the Factory to create a ResultSet object based on a Dynamic Query.

```
Set rs = objFact.DynamicSQL("SELECT Name, Price FROM MenuItem WHERE
                             Price > ? AND Name %startswith '?'")
rs.Execute 50, "SM"
```

You can use the CacheList control, which has a Factory property, and DynamicSQL and Run methods.

```
CacheList.Factory = objFact
CacheList.DynamicSQL ("SELECT Name, Price FROM MenuItem WHERE Price >
                       ? AND Name %startswith '?'")
CacheList.Run 50, "SM"
```

Form Wizard

The VB Form Wizard creates simple forms based on a class definition.

Most of the code generated by the Wizard is VB-specific.

Caché code is in the following subroutines:

Form_Load: creates the Factory and connects to Caché

action* subroutines (for buttons and menu): call sys_Save, sys_Close, sys_DeleteId, OpenId, New, and CacheQuery.FindId

SyncObjectToScreen: copies object's data to screen after OpenId or New

SyncScreenToObject: copies screen's data to object prior to Save

EXERCISES: MODULE 13

EXERCISES: MODULE 13

In this exercise, you create a VB form that allows you to edit Customers, and make some modifications.

1. Create a new Visual Basic project by selecting Standard EXE.

2. Use the Caché Form Wizard (under Add-Ins on the Menu) to create a form based on the Customer class, using all the properties except: LastName, Notes and Pets. Arrange the order of properties as you did in Module 12. Note: in order to choose the properties of the Home and Work Addresses, you must expand HomeAdd and WorkAdd and select the individual properties.

3. Using Project->References, add CacheObject.

4. From the Menu, select View->Code. Modify the declarations code at the top so that m_factory is declared as CacheObject.Factory.

5. Add a declaration of m_static as Object to the declarations code.

6. Use Edit->Replace to change every *m_object* to *cust*. Find the left-hand drop-down list at the top of the editing window and choose General, and in the right-hand drop-down list, choose ActionExit. Modify actionExit by replacing *actionClose* with *Unload Me*.

7. Using the left-hand drop-down list, select *Form*, which takes you to the Form_Load() event code. Modify that code so that it connects directly to your namespace, without using the Connection dialog box, using the following code:

```
        Set m_factory = New CacheObject.Factory
        con =
   m_factory.Connect("cn_iptcp:127.0.0.1[1972]:<namespa ce>")
        If Not con Then
            MsgBox "Cannot connect to Cache."
            End
        End If
```

8. To create a Form_Unload event, use the right-hand drop-down box and select Unload. Add this code:

```
        actionClose
        m_factory.Disconnect
        End
```

9. With General in the left-hand drop-down box, select *SyncObjectToScreen* on the right. Modify the SyncObjectToScreen subroutine to use LogicalToDisplay on the CustID property.:

```
        txtCustID.Text = cust.CustIDLogicalToDisplay(cust.CustID)
```

10. Modify the *SyncScreenToObject* subroutine to use DisplayToLogical on the CustID text field

```
cust.CustID = cust.CustIDDisplayToLogical(txtCustID.Text)
```

11. Return to View->Object. Use Tools->Menu Editor and Insert to add these 3 items following those already listed. Place the cursor at the end of the selections, and then fill in the boxes which should now be empty.

Caption	Name	Indentation
Populate	mnuPop	Left margin (copy format from &Object)
Add 50 Entries	mnuAdd	Use the->to ident (copy format from &New)
Delete All Entries	mnuDel	Use the->to ident (copy format from &New)

12. From the Form's menu, select *Add 50 Entries* to return to the code for this choice. Code the menu item that Adds 50 Entries to call the Populate() class method passing 50. Use the m_static object you declared earlier.

```
set m_static = m_factory.Static(m_classname)
m_static.Populate 50
set m_static = Nothing
```

13. Code the menu item that Deletes All Entries to call the %KillExtent() method, using the m_static object you declared earlier.

```
set m_static = m_factory.Static(m_classname)
m_static.sys_KillExtent
set m_static = Nothing
```

14. Save your project in *d:\mywork\vb*.

15. Run your form. Add, lookup, and edit entries. Check all fields with good and bad input. Test the menu items also.