



Caché Programming Orientation Guide

Version 2018.1
2024-04-03

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

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.

For Support questions about any InterSystems products, contact:

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

Table of Contents

About This Book	1
1 Introduction to Caché Programming	3
1.1 Introduction	3
1.2 Routines	4
1.3 Classes	5
1.4 Introduction to Globals	6
1.4.1 Ways to Access Data	7
1.4.2 Implications of Using Globals	7
1.5 Caché SQL	7
1.6 Macros	8
1.7 Include Files	8
1.8 How These Code Elements Work Together	9
2 A Closer Look at ObjectScript	11
2.1 Routines	11
2.2 Procedures, Functions, and Subroutines	14
2.3 Variables	15
2.3.1 Variable Names	16
2.3.2 Variable Types	16
2.3.3 Variable Length	17
2.3.4 Variable Existence	17
2.4 Variable Availability and Scope	17
2.4.1 Summary of Variable Scope	18
2.4.2 The NEW Command	18
2.5 Multidimensional Arrays	19
2.5.1 Basics	19
2.5.2 Structure Variations	20
2.5.3 Use Notes	20
2.6 Operators	21
2.6.1 Familiar Operators	21
2.6.2 Unfamiliar Operators	21
2.7 Commands	22
2.7.1 Familiar Commands	22
2.7.2 Commands for Use with Multidimensional Arrays	23
2.8 Special Variables	24
2.8.1 \$SYSTEM Special Variable	24
2.9 Locking and Concurrency Control	25
2.9.1 Basics	25
2.9.2 The Lock Table	26
2.9.3 Locks and Arrays	26
2.9.4 Introduction to Lock Types	27
2.10 System Functions	27
2.10.1 Value Choice	27
2.10.2 Existence Functions	28
2.10.3 List Functions	28
2.10.4 String Functions	28
2.10.5 Working with Multidimensional Arrays	29

2.10.6 Character Values	29
2.10.7 \$ZU Functions	30
2.11 Date and Time Values	30
2.11.1 Local Time	30
2.11.2 UTC Time	30
2.11.3 Date and Time Conversions	30
2.11.4 Details of the \$H Format	31
2.12 Using Macros and Include Files	31
2.13 Using Routines and Subroutines	31
2.13.1 Passing Variables by Value or by Reference	32
2.14 Potential Pitfalls	33
2.15 For More Information	35
3 Basic Ideas in Class Programming	37
3.1 Objects and Properties	37
3.2 Methods	38
3.2.1 Instance Methods	38
3.2.2 Class Methods	39
3.2.3 Methods and Variable Scope	39
3.3 Class Constants (Parameters)	40
3.4 Class Definitions and Types	40
3.5 Inheritance	41
3.5.1 Terminology and Basics	41
3.5.2 Example	42
3.5.3 Use of Inherited Class Members	42
3.5.4 Use of Subclasses	43
3.6 Classes as Containers of Methods	43
3.7 Abstract Classes	44
4 Caché Classes	45
4.1 Class Names and Packages	45
4.2 Basic Contents of a Class Definition	46
4.3 Class Name Shortcuts	48
4.4 Class Parameters	49
4.5 Properties	49
4.5.1 Specifying Property Keywords	50
4.6 Properties Based on Data Types	51
4.6.1 Data Type Classes	51
4.6.2 Overriding Parameters of Data Type Classes	52
4.6.3 Using Instance Variables	53
4.6.4 Using Other Property Methods	53
4.7 Methods	53
4.7.1 Specifying Method Keywords	53
4.7.2 References to Other Class Members	54
4.7.3 References to Methods of Other Classes	55
4.7.4 References to Current Instance	55
4.7.5 Method Arguments	56
4.8 Special Kinds of Methods	57
4.8.1 Call Methods	57
4.8.2 Method Generators	58
4.9 Class Queries	58
4.10 XData Blocks	58

4.11 Macros and Include Files in Class Definitions	59
4.12 Inheritance Rules in Caché	59
4.12.1 Inheritance Order	59
4.12.2 Primary Superclass	59
4.12.3 Most-Specific Type Class	60
4.12.4 Overriding Methods	60
4.13 For More Information	60
5 Caché Objects	61
5.1 Introduction to Caché Object Classes	61
5.2 Basic Features of Object Classes	62
5.3 OREFs	64
5.4 Stream Interface Classes	65
5.4.1 Stream Classes	65
5.4.2 Example	65
5.5 Collection Classes	65
5.5.1 List and Array Classes for Use As Standalone Objects	66
5.5.2 List and Arrays as Properties	66
5.6 Useful ObjectScript Functions	67
5.7 For More Information	67
6 Persistent Objects and Caché SQL	69
6.1 Introduction	69
6.2 Caché SQL	70
6.2.1 Where You Can Use Caché SQL	70
6.2.2 Object Extensions to SQL	70
6.3 Special Options for Persistent Classes	71
6.4 SQL Projection of Persistent Classes	72
6.4.1 Demonstration of the Object-SQL Projection	72
6.4.2 Basics of the Object-SQL Projection	72
6.4.3 Classes and Extents	73
6.5 Object IDs	73
6.5.1 How an ID Is Determined	74
6.5.2 Accessing an ID	74
6.6 Storage	75
6.6.1 A Look at a Storage Definition	75
6.6.2 Globals Used by a Persistent Class	75
6.6.3 Default Structure for a Stored Object	76
6.6.4 Notes	76
6.7 Options for Creating Persistent Classes and Tables	76
6.8 Accessing Data	77
6.9 A Look at Stored Data	77
6.10 Storage of Generated Code for Caché SQL	79
6.11 For More Information	79
7 Namespaces and Databases	81
7.1 Introduction to Namespaces and Databases	81
7.1.1 Locks, Globals, and Namespaces	82
7.2 Database Basics	82
7.2.1 Database Configuration	82
7.2.2 Database Features	83
7.2.3 Database Portability	83

7.3 System-Supplied Databases	84
7.4 System-Supplied Namespaces	85
7.5 Custom Items in CACHESYS	85
7.6 What Is Accessible in Your Namespaces	86
7.6.1 System Globals in Your Namespaces	86
7.7 Stream Directory	86
7.8 For More Information	87
8 Caché Security	89
8.1 Introduction	89
8.1.1 Security Elements Within Caché	89
8.1.2 Secure Communications to and From Caché	90
8.2 Caché Applications	90
8.3 Caché Authorization Model	91
9 Localization Support	93
9.1 Introduction	93
9.2 Caché Locales and National Language Support	93
9.3 Default I/O Tables	94
9.4 Files and Character Encoding	95
9.5 Manually Translating Characters	95
10 Development Tools	97
10.1 Management Portal	97
10.1.1 Accessing the Portal	97
10.2 Studio	98
10.2.1 Starting Studio	98
10.2.2 Orientation	98
10.2.3 Documents (“Files”)	98
10.2.4 Integration with Source Control	99
10.2.5 Other Environmental Options	99
10.3 Terminal	99
10.3.1 Starting the Terminal	99
10.3.2 Orientation	100
10.3.3 Environmental Options	100
10.3.4 Using the Terminal	101
10.3.5 Common Keyboard Accelerators	101
10.4 System Qualifiers and Flags	101
10.4.1 Viewing the Current Default Qualifiers and Flags	102
10.4.2 Changing the Defaults	102
10.4.3 Key Flags	102
10.4.4 Historical Note	102
10.5 InterSystems Classes and Routines	103
10.6 InterSystems Class Reference (Documatic)	103
10.6.1 Accessing the InterSystems Class Reference	103
10.6.2 A Quick Look at the InterSystems Class Reference	103
10.7 Tools for Debugging	105
10.7.1 Studio Debugger	106
10.7.2 ZBREAK	106
10.7.3 ^%STACK	106
10.7.4 Trace Statements	107
10.7.5 Audit Log	107

10.7.6 System Logs	107
10.8 For More Information	107
11 Server Configuration Options	109
11.1 Support for Long String Operations	109
11.1.1 Enabling Long String Operations	110
11.2 Settings for Caché SQL	110
11.3 Use of IPv6 Addressing	110
11.4 Configuring a Server Programmatically	111
11.5 For More Information	111
12 Useful Skills to Learn	113
12.1 Finding Definitions of Code Elements	114
12.1.1 Finding a Class Member in Studio	114
12.1.2 Finding a Class Member in the InterSystems Class Reference	114
12.1.3 Finding Subclasses in the InterSystems Class Reference	114
12.1.4 Finding a Macro in Studio	115
12.2 Defining Databases	115
12.3 Defining Namespaces	115
12.4 Mapping a Global	116
12.5 Mapping a Routine	117
12.6 Mapping a Package	118
12.7 Generating Test Data	119
12.7.1 Extending %Populate	119
12.7.2 Using Methods of %Populate and %PopulateUtils	120
12.8 Removing Stored Data	120
12.9 Resetting Storage	121
12.10 Browsing a Table	122
12.11 Executing an SQL Query	123
12.12 Examining Object Properties	123
12.13 Viewing Globals	124
12.14 Displaying INT Code	125
12.15 Testing a Query and Viewing a Query Plan	126
12.16 Viewing the Query Cache	127
12.17 Building an Index	127
12.18 Using the Tune Table Facility	128
12.19 Moving Code from One Database to Another	128
12.20 Moving Data from One Database to Another	129
12.21 Renaming a Class	130
12.22 For More Information	130
Appendix A: What's That?	131
A.1 Non-Alphanumeric Characters in the Middle of "Words"	131
A.2 . (One Period)	133
A.3 .. (Two Periods)	134
A.4 # (Pound Sign)	134
A.5 Dollar Sign (\$)	135
A.6 Percent Sign (%)	136
A.7 Caret (^)	137
A.8 Other Forms	138
Appendix B: Rules and Guidelines for Identifiers	141
B.1 Namespaces	141

B.1.1 Namespace Names to Avoid	141
B.2 Databases	142
B.2.1 Database Names to Avoid	142
B.3 Local Variables	142
B.3.1 Local Variable Names to Avoid	142
B.4 Global Variables	142
B.4.1 Global Variable Names to Avoid	143
B.5 Routines and Labels	145
B.5.1 Reserved Routine Names for Your Use	145
B.6 Classes	146
B.6.1 Class Names to Avoid	146
B.7 Class Members	146
B.7.1 Member Names to Avoid	147
B.8 Custom Items in CACHESYS	147
Appendix C: General System Limits	149
C.1 Long String Limit	149
C.2 Class Limits	150
C.3 Class and Routine Limits	151
C.4 Other Programming Limits	153
Appendix D: Numeric Computing in InterSystems Applications	155
D.1 Representations of Numbers	155
D.1.1 Decimal Format	155
D.1.2 \$DOUBLE Format	156
D.1.3 SQL Representations	157
D.2 Choosing a Numeric Format	157
D.3 Converting Numeric Representations	158
D.3.1 Strings	158
D.3.2 Decimal to \$DOUBLE	159
D.3.3 \$DOUBLE to Decimal	159
D.3.4 Decimal to String	160
D.4 Operations Involving Numbers	160
D.4.1 Arithmetic	160
D.4.2 Comparison	161
D.4.3 Boolean Operations	162
D.5 Summary of Changes Introduced in Version 2008.2	162
D.6 See Also	162

About This Book

This book is an orientation guide for programmers who are new to Caché or who are familiar with only some kinds of Caché programming. It focuses on server-side programming.

It consists of the following topics:

- [Introduction to Caché Programming](#)
- [A Closer Look at ObjectScript](#)
- [Basic Ideas in Class Programming](#)
- [Caché Classes](#)
- [Caché Objects](#)
- [Persistent Objects and Caché SQL](#)
- [Namespaces and Databases](#)
- [Caché Security](#)
- [Localization Support](#)
- [Development Tools](#)
- [Server Configuration Options](#)
- [Useful Skills to Learn](#)
- [What's That?](#)
- [Rules and Guidelines for Identifiers](#)
- [General System Limits](#)
- [Numeric Computing in InterSystems Applications](#)

For a detailed outline, see the [table of contents](#).

For information about related topics, see the following documents:

- [Using Caché Objects](#) describes how to create and work with classes and objects.
- [Using Caché ObjectScript](#) describes concepts and how to use the ObjectScript language.
- [Using Caché SQL](#) describes how to use Caché SQL and where you can use this language.
- [Using Caché Globals](#) describes how to work with multidimensional arrays and globals.
- [InterSystems Programming Tools Index](#) is a directory of the InterSystems classes, routines, bindings, and other programming tools, grouped by topic.

For general information, see [Using InterSystems Documentation](#).

1

Introduction to Caché Programming

This book is intended as an introduction for programmers who are not familiar with Caché or who are familiar with only some kinds of Caché programming. It is not a tutorial but rather a survey of the elements in the Caché toolkit, with information on how these elements fit together. After reading this book, you should have an idea of the options available to you and where to find more information on those options.

For this book, the emphasis is on server-side programming, rather than client-side programming. Via various Caché language bindings, a Caché server can work with clients written in many different languages. If you are creating or maintaining such clients, most of this book is probably inapplicable to you.

This chapter provides a high-level overview of the language elements you can use in Caché server-side programs and shows how Caché combines them. It discusses the following topics:

- [Introduction](#)
- [Routines](#)
- [Classes](#)
- [Globals](#)
- [Caché SQL](#)
- [Macros](#)
- [Include files](#)
- [How these elements work together](#)

1.1 Introduction

Caché is a high-performance object database with several built-in general-purpose programming languages. It supports multiple processes and provides concurrency control. Each process has direct, efficient access to the data.

In Caché, you can write routines, classes, or a mix of these, as suits your preferences and the history of your application. In all cases, stored data is ultimately contained in structures known as globals (a later section in this chapter discusses these). Caché programming has the following features:

- Routines and classes can be used interchangeably and can be written in more than one language.
- Routines and classes can call each other.
- Classes provide object-oriented features.

- Database storage is an integrated part of all Cache programming languages.
- Classes can persist data in a way that simplifies programming. If you use persistent classes, data is simultaneously available as objects, SQL tables, and globals.
- You can access globals directly from either routines or classes, which means that you have the flexibility to store and access data exactly how you want.

Older Caché applications consist entirely of routines, because these applications were written before Caché supported class definitions. In contrast, some newer applications are written almost entirely in classes. You can choose the approach that is appropriate for your needs.

1.2 Routines

When you create routines in Caché, you can choose the programming language for each routine. The choices are as follows:

- ObjectScript, which is a superset of the ISO 11756-1999 standard M programming language. If you are an M programmer, you can run your existing M applications on Caché with no change.

This is the most common language for Caché routines.

- Caché MVBasic is an implementation of [MultiValue](#). It includes commands, functions, and operators that are used in the various MultiValue implementations, and it supports multiple emulation modes so that you can use syntax that is familiar to you.
- Caché Basic is an implementation of Basic.

Because it is useful to see what these languages look like, the following shows part of a routine written in ObjectScript:

```
SET text = ""
FOR i=1:5:$LISTLENGTH(attrs)
{
    IF ($ZCONVERT($LIST(attrs, (i + 1)), "U") = "XREFLABEL")
    {
        SET text = $LIST(attrs, (i + 4))
        QUIT
    }
}

IF (text = "")
{
    QUIT $$$ERROR($$$GeneralError,$$$(T("Missing xreflabel value"))
}
```

The following shows part of a routine written in MVBasic:

```
#PRAGMA ROUTINENAME=ADDRROUTINENAME
$OPTIONS CACHE
OPEN 'VOC' TO F.VOC ELSE STOP
BAD.LST = ''
* If there is an active select list use it, otherwise get file names from the VOC
IF SYSTEM(11) > 0 THEN
    NBR.PROG.FILES = SYSTEM(11)
END ELSE
    EXECUTE 'SELECT VOC WITH F6 LIKE B...' CAPTURING RES RETURNING NBR.PROG.FILES
END
IF NBR.PROG.FILES THEN
    CRT 'THERE ARE ':NBR.PROG.FILES:' PROGRAM FILES'
    EXECUTE 'SAVE-LIST PGM.FILES' PASSLIST
    READLIST ALL.PROG.FILES FROM 'PGM.FILES' ELSE STOP
    HOW.MANY = DCOUNT(ALL.PROG.FILES,@AM)
    FOR I = 1 TO HOW.MANY
        THIS.PROG.FILE = ALL.PROG.FILES<I>
        CRT
    ...
```

The following shows part of a routine written in Caché Basic:

```
' display an ordered list of matches
' user can enter full or partial name, full or partial phone, or a valid date
' pick from a list of matches, and EDIT their choice
Option Explicit
dim id, name, phone, intdob, matches

public sub main()
    dim done
    do
        getsubmit(id, done) ' let user submit a string for lookup
        if id = 0 then continue do
        display(id, "table") ' display the chosen person
        edit(id) ' edit the chosen person
    loop until done
end sub

private sub getsubmit(ByRef id as %Integer, ByRef done as %Boolean)
' ask user what to search for, and take appropriate action
    dim submit
    id = 0 : done = False
    println : input "Lookup: ", submit : println
    if (submit = "") then ' user entered nothing
        done = True
        exit sub
    end if
' figure out what user entered
    ...
```

The [next chapter](#) provides an introduction to ObjectScript. This book does not discuss Caché MVBasic or Caché Basic in any detail.

1.3 Classes

Caché also supports classes. You can use the system classes and you can define your own classes.

In Caché, a class can include familiar class elements such as properties, methods, and parameters (known as constants in other class languages). It can also include items not usually defined in classes, including triggers, queries, and indices.

The following shows a class definition:

Class Definition

```
Class Sample.Employee Extends Person
{

    /// The employee's job title.
    Property Title As %String(MAXLEN = 50);

    /// The employee's current salary.
    Property Salary As %Integer(MAXVAL = 100000, MINVAL = 0);

    /// A character stream containing notes about this employee.
    Property Notes As %Stream.GlobalCharacter;

    /// A picture of the employee
    Property Picture As %Stream.GlobalBinary;

    /// This method overrides the method in the Person class.
    Method PrintPerson()
    {
        Write !,"Name: ", ..Name, ?30, "Title: ", ..Title
        Quit
    }
}
```

For each method, you can specify the programming language to use in the definition of that method, although typically all methods in a class use the same language, for simplicity. The default is ObjectScript; the [next chapter](#) provides an introduction

to it. The other choices are Caché MVBasic, Caché Basic, and Transact-SQL. (There are additional options for client-side programming, not discussed in this book.)

You can use classes from within routines. For example, the following shows part of a routine, in which we refer to the `Sample.Person` class:

ObjectScript

```
//get details of requested person; print them
showperson() public {
    set rand=$RANDOM(10)+1      ; rand is an integer in the range 1-10
    write "Your random number: " _rand
    set person=##class(Sample.Person).%OpenId(rand)
    write !,"This person's name: " _person.Name
    write !,"This person's age: " _person.Age
    write !,"This person's home city: " _person.Home.City
}
```

Similarly, a method can invoke a label in a routine. For example:

Class Member

```
Method DemoRoutineCall(input) as %String
{
    Set value=$$function^myroutine(input)
    Quit value
}
```

The chapter “[Basic Ideas in Class Programming](#)” provides a brief introduction to class programming, for the benefit of readers who have not done this kind of programming. The chapters after that discuss [classes in Caché](#) and the unique capabilities of [persistent classes](#) in Caché.

1.4 Introduction to Globals

Caché supports a special kind of variable that is not seen in other programming languages; this is a *global variable*, which is usually just called a *global*. In Caché, the term *global* indicates that this data is available to all processes accessing this database. This usage is different from other programming languages in which *global* means “available to all code in this module.”

The contents of a global are stored in a Caché database. The [next chapter](#) introduces these more thoroughly; for now, it is important just to know the following points:

- A global consists of a set of nodes (in some cases, only one node), identified by subscripts.
Each node can contain a value.
- ObjectScript, Caché Basic, and Caché MVBasic include functions to iterate through the nodes of a global and quickly access values.
- A global is automatically stored in the database. When you assign a value to a node of a global variable, the data is written immediately to the database.
- You can see the contents of a global via an ObjectScript command or via the Management Portal.

1.4.1 Ways to Access Data

In Caché, a database contains globals and nothing else; even code is stored in globals, as described later in this book. At the lowest level, all access to data is done via *direct global access* — that is, by using commands and functions that work directly with globals.

Many currently running applications were developed long before Caché included support for classes. Some of these applications use direct global access. Other applications use custom APIs such as FileMan, which is in the public domain. These APIs, of course, internally use direct global access.

When you use persistent classes, discussed later in this book, you can create, modify, and delete stored data in either of the following ways:

- By using methods such as `%New()`, `%Save()`, `%Open()`, and `%Delete()`.
- By using Caché SQL.

Internally, the system always uses direct global access.

Because object classes provide a more controlled interface, and because Caché persistent classes are projected to tables that can be queried via SQL, it is often desirable to add a class interface to existing applications. You can do so, if you understand the structure of the globals.

1.4.2 Implications of Using Globals

Globals are stored physically in a highly optimized structure, and the code that manages this structure is separately optimized for every platform that Caché runs on. These optimizations ensure that operations on globals have high throughput (number of operations per unit of time), high concurrency (total number of simultaneous users), efficient use of cache memory, and require no ongoing performance-related maintenance (such as frequent rebuilding, re-indexing, or compaction). The physical structure used to store globals is completely encapsulated; it is generally unnecessary to consider the physical data structure.

Global storage is sparse, meaning that only nodes with data values are stored in the database. This means that Caché often requires less than half of the space needed by a relational database.

Other benefits of globals include the following:

- The hierarchical structure of globals typically models real-world data more closely than is possible with relational tables.
- The sparse nature means that new fields can be added without any overhead and without rebuilding the existing database.
- More data can be read or written with a single I/O operation, and data can be cached more efficiently.

1.5 Caché SQL

As noted previously, Caché provides an implementation of SQL, known as Caché SQL.

You can use Caché SQL within routines and within methods. To use SQL in these contexts, you can use either or both of the following tools:

- *Dynamic SQL* (the `%SQL.Statement` and `%SQL.StatementResult` classes), as in the following example:

ObjectScript

```
SET myquery = "SELECT TOP 5 Name,Home_City FROM Sample.Person ORDER BY Age"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatus = tStatement.%Prepare(myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

You can use dynamic SQL in any context.

- *Embedded SQL*, as in the following example:

ObjectScript

```
&sql(SELECT COUNT(*) INTO :myvar FROM Sample.Person)
Write myvar
```

The first line is embedded SQL, which executes a Caché SQL query and writes a value into a *host variable* called `myvar`.

The next line is ordinary ObjectScript; it simply writes the value of the variable `myvar`.

You can use embedded SQL in ObjectScript routines and in methods written in ObjectScript.

1.6 Macros

ObjectScript also supports *macros*, which define substitutions. The definition can either be a value, an entire line of code, or (with the `##continue` directive) multiple lines. You use macros to ensure consistency. For example:

ObjectScript

```
#define StringMacro "Hello, World!"
write $$$StringMacro
```

To give you an idea of what can be done in macros, the following example shows the definition of a macro that is used internally:

ObjectScript

```
#define output1(%str,%lf,%indent) do output^%fm2class(%str,%lf,%indent,$$$display)
```

This macro accepts an argument, as many of them do. It also refers to another macro.

You can use macros in routines as well as in classes. Some of the system classes use them extensively.

1.7 Include Files

You can define macros in a routine and use them later in the same routine. More commonly, you define them in a central place. To do this, you create and use *include files*. An include file defines macros and can include other include files.

The following shows parts of a system include file:

ObjectScript

```

/// Create a success %Status code
#define OK 1

/// Return true if the %Status code is success, and false otherwise
/// %sc - %Status code
#define ISOK(%sc) (+%sc)

/// Return true if the %Status code if an error, and false otherwise
/// %sc - %Status code
#define ISERR(%sc) ('%sc)

```

Here is another include file, in entirety:

ObjectScript

```

#include %occCacheDirect
#include %occExtent
#include %occTransaction
#include %occInclude
#include %msql
#include %cspInclude

```

Then you can do the following:

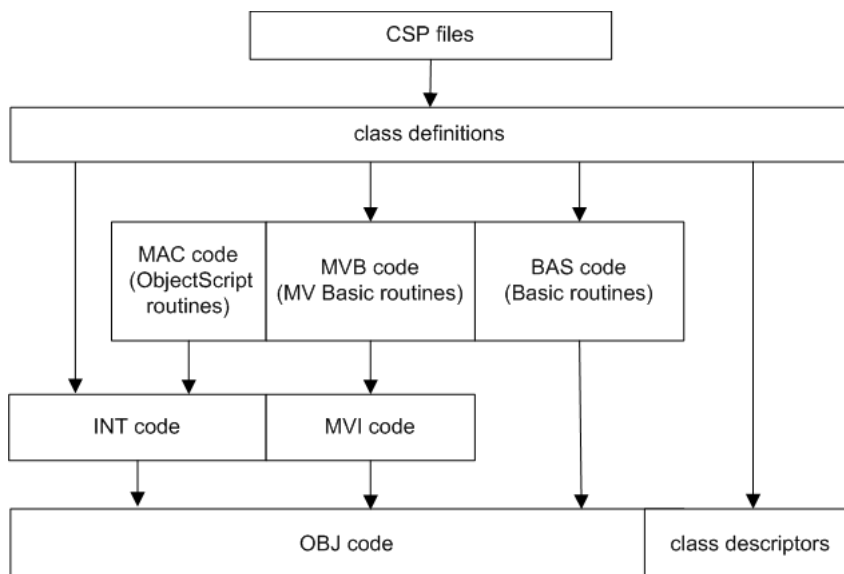
- Include the include file at the start of any routine. That routine can refer to the macros defined in the include file.
- Include the include file at the start of any class. Methods in that class can refer to the macros.
- Include the include file at the start of any method. That method can refer to the macros.

Note that you use slightly different syntax to include an include file within a class definition. See “[Macros and Include Files in Class Definitions](#),” later in this book.

1.8 How These Code Elements Work Together

It is useful to understand how Caché uses the code elements introduced in this chapter.

The reason that you can use a mix of ObjectScript, Caché SQL, Caché MVBasic, macros, class definitions, routines, and so on is that Caché does not *directly* use the code that you write. Instead, when you compile your code, the system generates the code that it uses, which is OBJ code, used by the Caché virtual machine.



There are multiple steps; the preceding figure gives a general idea of them. It is not necessary to know the steps in detail, but the following points are good to remember:

- The *CSP engine* converts CSP files into class definitions.
- The *class compiler* uses the class definitions and generates INT code, Caché MVBasic code, and Caché Basic code, depending on the languages used to define the methods in the classes.

In some cases, the compiler first uses classes as the basis for generating and saving additional classes. You can look at these classes in Studio, but do not modify them. This occurs, for example, when you compile classes that define web services and web clients.

The class compiler also generates the class descriptor for each class. The system code uses this at runtime.

- A *preprocessor* (sometimes called the *macro preprocessor* or *MPP*) uses the include files (not shown in the figure) and replaces the macros. It also handles the embedded SQL in the ObjectScript routines and the MVBasic routines.

These changes occur in a temporary work area, and your code is not changed.

- Additional compilers create INT code for the ObjectScript routines and MVI code for the MVBasic routines. This layer is known as intermediate code. In this layer, all access to the data is done via direct global access.

Both INT code and MVI code are compact but human-readable. A later section of this book shows you how to find this code, which can be useful for diagnostic purposes.

Note that there is no intermediate code for Caché Basic.

- INT code, MVI code, and Caché Basic are used to generate OBJ code.

The Caché virtual machine uses this code. Once you have compiled your code, the routines, INT code, and MVI code are no longer necessary for code execution.

- After you compile your classes, you can put them into *deployed* mode. Caché has a utility that removes the class internals and the intermediate code for a given class; you can use this utility when you deploy your application.

If you examine the Caché system classes, you might find that some classes cannot be seen because they are in deployed mode.

Note: All routines and class definitions are stored in the same Caché databases as the generated code. This fact makes the code easier to manage. Caché provides a robust set of source control hooks for Studio that InterSystems developers have used for many years. You can use these hooks as well; for an introduction, see “[Development Tools](#),” later in this book.

2

A Closer Look at ObjectScript

Most methods and most routines are written in ObjectScript. This chapter gives an overview of this language, if you intend to use it or if you need to understand code that other people have written. This chapter discusses the following topics:

- [Routines](#)
- [Procedures, functions, and subroutines](#)
- [Variables](#)
- [Variable availability and scope](#)
- [Multidimensional arrays](#)
- [Operators](#)
- [Commands](#)
- [Special variables](#)
- [Locking and concurrency control](#)
- [Functions](#)
- [Date and time values](#)
- [Macros and include files](#)
- [How to use routines, procedures, functions, and subroutines](#)
- [Potential pitfalls](#)
- [Other sources of information on these topics](#)

A method can contain the same statements, labels, and comments as routines do. That is, all the information here about the contents of a routine also applies to the contents of a method.

If you are writing classes rather than routines, start with the section “[Variables](#).”

2.1 Routines

The following shows an example routine named `demoroutine` that is written in ObjectScript. This example gives us an opportunity to see some common commands, operators, and functions, and to see how code is organized within a routine.

ObjectScript

```
; this is demoroutine
write "Use one of the following entry points:"
write !,"random"
write !,"input"
write !,"interesting"
quit

//this procedure can be called from outside the routine
random() public {
    set rand=$RANDOM(10)+1      ; rand is an integer in the range 1-10
    write "Your random number: "_rand
    set name=$$getnumbername(rand)
    write !, "Name of this number: "_name
}

//this procedure can be called from outside the routine
input() public {
    read "Enter a number from 1 to 10: ", input
    set name=$$getnumbername(input)
    write !, "Name of this number: "_name
}

//this procedure can be called only from within this routine
getnumbername(number) {
    set name=$CASE(number,1:"one",2:"two",3:"three",
        4:"four",5:"five",6:"six",7:"seven",8:"eight",
        9:"nine",10:"ten",:"other")
    quit name
}

/* write some interesting values
this procedure can be called from outside the routine
*/
interesting() public {
    write "Today's date: "_$ZDATE($HOROLOG,3)
    write !,"Your installed version: "_$ZVERSION
    write !,"Your username: "_$USERNAME
    write !,"Your security roles: "_$ROLES
}
```

Note the following highlights:

- The only identifier that actually starts with a caret (^) is the name of a global; these are discussed later in this chapter. However, in running text and in code comments, it is customary to refer to a routine as if its name started with a caret, because you use the caret when you invoke the routine (as shown later in this chapter). For example, the routine `demoroutine` is usually called `^demoroutine`.
- The routine name does not have to be included within the routine. When you view a routine in Studio, the routine name is displayed on the tab that displays the routine. For example:

```

demoroutine.mac

; this is demoroutine
write "Use one of the following entry points:"
write !, "random"
write !, "input"
write !, "interesting"
quit

//this procedure can be called from outside the routine
random() public {
    set rand=$RANDOM(10)+1          ; rand is an integer in the
    write "Your random number: "_rand
    set name=$$getnumbername(rand)
    write !, "Name of this number: "_name
}

//this procedure can be called from outside the routine
input() public {
    read "Enter a number from 1 to 10: ", input

```

Many programmers include the routine name as a comment at the start of the routine or as the first label in the routine.

- The routine has multiple *labels*: random, input, getnumbername, and interesting.

Labels are used to indicate the starting point for procedures (as in this example), functions, and subroutines; these terms are defined [later in this chapter](#). You can also use them as a destination for certain commands.

Labels are common in routines, but you can also use them within methods.

Labels are also called *entry points* or *tags*.

- The random and input subroutines invoke the getnumbername subroutine.
- **WRITE**, **QUIT**, **SET**, and **READ** are commands. The language includes other commands to remove variables, commands to control program flow, commands to control I/O devices, commands to manage transactions (possibly nested), and so on.

The names of commands are not case-sensitive, although they are shown in running text in all upper case by convention.

- The sample includes two operators. The plus sign (+) performs addition, and the underscore (_) performs string concatenation.

ObjectScript provides the usual operators and some special operators not seen in other languages.

- **\$RANDOM**, **\$CASE**, and **\$ZDATE** are functions.

The language provides functions for string operations, conversions of many kinds, formatting operations, mathematical operations, and others.

- **\$HOROLOG**, **\$ZVERSION**, **\$USERNAME**, and **\$ROLES** are system variables (called *special variables* in Caché). Most special variables contain values for aspects of the Caché operating environment, the current processing state, and so on.
- ObjectScript supports comment lines, block comments, and comments at the end of statements.

We can execute parts of this routine in the Terminal, as a demonstration. First, the following shows a Terminal session, in which we run the routine itself. In these examples, `SAMPLES>` is the prompt shown in the Terminal. The text after the

prompt on the same line is the entered command. The lines after that show the values that the system writes to the Terminal in response.

```
SAMPLES>do ^demoroutine
Use one of the following entry points:
random
input
SAMPLES>
```

When we run the routine, we just get help information, as you can see. It is not required to write your routines in this way, but it is common. Note that the routine includes a **QUIT** before the first label, to ensure that when a user invokes the routine, processing is halted before that label. This practice is also not required, but is also common.

Next, the following shows how a couple of the subroutines behave:

```
SAMPLES>do input^demoroutine
Enter a number from 1 to 10: -7
Name of this number: other
SAMPLES>do interesting^demoroutine
Today's date: 2010-11-30
Your installed version: Cache for Windows (x86-32) 2010.2 (Build 454U) Sun Oct 24 2010 17:14:03 EDT
Your username: UnknownUser
Your security roles: %All
SAMPLES>
```

2.2 Procedures, Functions, and Subroutines

Within an ObjectScript routine, a label defines the starting point for one of the following units of code:

- *Procedure* (optionally returns a value). The variables defined in a procedure are private to that procedure, which means that they are not available to other code. This is not true for functions and subroutines.

A procedure is also called a *procedure block*.

- *Function* (returns a value).
- *Subroutine* (does not return a value).

InterSystems recommends that you use procedures, because this simplifies the task of controlling the scope of variables. In existing code, however, you might also see functions and subroutines, and it is useful to be able to recognize them. The following list shows what all these forms of code look like.

procedure

```
label(args) scopekeyword {
    zero or more lines of code
    QUIT returnvalue
}
```

Or:

```
label(args) scopekeyword {
    zero or more lines of code
}
```

label is the identifier for the procedure.

args is an optional comma-separated list of arguments. Even if there are no arguments, you must include the parentheses.

The optional *scopekeyword* is one of the following (not case-sensitive):

- `Public`. If you specify `Public`, then the procedure is *public* and can be invoked outside of the routine itself.
- `Private` (the default for procedures). If you specify `Private`, the procedure is *private* and can be invoked only by other code in the same routine. If you attempt to access the procedure from another routine, a `<NOLINE>` error occurs.

returnvalue is an optional, single value to return. To return a value, you must use the **QUIT** command. If you do not want to return a value, you can omit the **QUIT** command, because the curly braces indicate the end of the procedure.

A procedure can declare variables as public variables, although this practice is not considered modern. To do this, you include a comma-separated list of variable names in square brackets immediately before *scopekeyword*. For details, see “[User-defined Code](#)” in *Using Caché ObjectScript*.

function

```
label(args) scopekeyword
    zero or more lines of code
QUIT optionalreturnvalue
```

args is an optional comma-separated list of arguments. Even if there are no arguments, you must include the parentheses.

The optional *scopekeyword* is either `Public` (the default for functions) or `Private`.

subroutine

```
label(args) scopekeyword
    zero or more lines of code
QUIT
```

args is an optional comma-separated list of arguments. If there are no arguments, the parentheses are optional.

The optional *scopekeyword* is either `Public` (the default for subroutines) or `Private`.

The following table summarizes the differences among routines, subroutines, functions, and procedures:

	Routine	Subroutine	Function	Procedure
Can accept arguments	no	yes	yes	yes
Can return a value	no	no	yes	yes
Can be invoked outside the routine (by default)	yes	yes	yes	no
Variables defined in it are available after the code finishes execution	yes	yes	yes	depends on nature of the variable

The section “[Variable Availability and Scope](#),” later in this chapter, has further details on variable scope.

Note: In casual usage, the term *subroutine* can mean procedure, function, or subroutine (as defined formally here).

2.3 Variables

In ObjectScript, there are two kinds of variables, as categorized by how they hold data:

- *Local variables*, which hold data in memory.

Local variables can have public or private scope.

- *Global variables*, which hold data in a database. These are also called *globals*. All interactions with a global affect the database immediately. For example, when you set the value of a global, that change immediately affects what is stored; there is no separate step for storing values. Similarly, when you remove a global, the data is immediately removed from the database.

There are special kinds of globals not discussed here; see “[Caret \(^\)](#)” in the appendix “[What’s That?](#)”

2.3.1 Variable Names

The names of variables follow these rules:

- For most local variables, the first character is a letter, and the rest of the characters are letters or numbers. Valid names include `myvar` and `i`
- For most global variables, the first character is always a caret (^). The rest of the characters are letters, numbers, or periods. Valid names include `^myvar` and `^my.var`

Caché also supports a special kind of variable known as a *percent variable*; these are less common. The name of a percent variable starts with a percent character (%). Percent variables are special in that they are always public; that is they are visible to all code within a process. This includes all methods and all procedures within the calling stack.

When you define percent variables, use the following rules:

- For a local percent variable, start the name with `%Z` or `%z`. Other names are reserved for system use.
- For a global percent variable, start the name with `^%Z` or `^%z`. Other names are reserved for system use.

For details on percent variables and variable scope, see “[Variable Availability and Scope](#),” later in this chapter; also see “[Callable User-defined Code Modules](#)” in *Using Caché ObjectScript*.

For further details on names and for variations, see “[Syntax Rules](#)” in *Using Caché ObjectScript*. Or see “[Rules and Guidelines for Identifiers](#),” later in this book.

2.3.2 Variable Types

Variables in ObjectScript are weakly, dynamically typed. They are dynamically typed because you do not have to declare the type for a variable, and variables can take any legal value — that is, any legal literal value or any legal ObjectScript expression. They are weakly typed because usage determines how they are evaluated.

A legal literal value in ObjectScript has one of the following forms:

- Numeric. Examples: `100`, `17.89`, and `1e3`
- Quoted string, which is a set of characters contained within a matched set of quotation marks ("). For example: `"my string"`

To include a double quote character within a string literal, precede it with another double quote character. For example: `"This string has ""quotes"" in it."`

Depending on the context, a string can be treated as a number and vice versa. Similarly, in some contexts, a value may be interpreted as a boolean (true or false) value; anything that evaluates to zero is treated as false; anything else is treated as true.

When you create classes, you can specify types for properties, for arguments to methods, and so on. The Caché class mechanisms enforce these types as you would expect. [A later section of this book](#) provides an overview of Caché data type classes.

2.3.3 Variable Length

There is a limit to the length of a value of a variable. If you have *long strings* enabled in your installation, the limit is 3,641,144 characters. If long strings are not enabled, the limit is 32,767 characters.

A later section of this book explains [how to enable long string operations](#).

2.3.4 Variable Existence

You usually define a variable with the **SET** command. As noted earlier, when you define a global variable, that immediately affects the database.

A global variable becomes undefined only when you *kill* it (which means to remove it via the **KILL** command). This also immediately affects the database.

A local variable can become undefined in one of three ways:

- It is killed.
- The process (in which it was defined) ends.
- It goes out of scope within that process.

To determine whether a variable is defined, you use the **\$DATA** function. For example, the following shows a Terminal session that uses this function:

```
SAMPLES>write $DATA(x)
0
SAMPLES>set x=5
SAMPLES>write $DATA(x)
1
```

In the first step, we use **\$DATA** to see if a variable is defined. The system displays 0, which means that the variable is not defined. Then we set the variable equal to 5 and try again. Now the function returns 1.

In this example and in previous examples, you may have noticed that it is not necessary to declare the variable in any way. The **SET** command is all that you need.

If you attempt to access an undefined variable, you get the <UNDEFINED> error. For example:

```
SAMPLES>WRITE testvar
WRITE testvar
^
<UNDEFINED> *testvar
```

2.4 Variable Availability and Scope

ObjectScript supports the following program flow, which is similar (in most ways) to what other programming languages support:

1. A user invokes a procedure, perhaps from a user interface.
2. The procedure executes some statements and then invokes another procedure.
3. The procedure defines local variables A, B, and C.

Variables A, B, and C are *in scope* within this procedure. They are *private* to this procedure.

The procedure also defines the global variable ^D.

4. The second procedure ends, and control returns to the first procedure.
5. The first procedure resumes execution. This procedure cannot use variables A, B, and C, which are no longer defined. It can use ^D, because that variable was immediately saved to the database.

The preceding program flow is quite common. Caché provides other options, however, of which you should be aware.

2.4.1 Summary of Variable Scope

Several factors control whether a variable is available outside of the unit of code that defines it. Before discussing those, it is necessary to point out the following environmental details:

- A Caché instance includes multiple namespaces, including multiple system namespaces and probably multiple namespaces that you define.

A namespace is the environment in which any code runs. Namespaces are discussed [later](#) in more detail.

- You can run multiple processes simultaneously in a namespace. In a typical application, many processes are running at the same time.

The following table summarizes where variables are available:

Variable availability, broken out by kind of variable...	Outside of unit of code that defines it (but in the same process)	In other processes in the same namespace	In other namespaces within same Caché instance
Local variable, private scope [*]	No	No	No
Local variable, public scope	Yes	No	No
Local percent variable	Yes	No	No
Global variable (not percent)	Yes	Yes	Not unless global mappings permit this†
Global percent variable	Yes	Yes	Yes

^{*}By default, variables defined in a procedure are private to the procedure, as noted before. Also, in a procedure, you can declare variables as public variables, although this practice is not preferred. See “[User-defined Code](#)” in *Using Caché ObjectScript*.

†Each namespace has default databases for specific purposes and can have mappings that give access to additional databases. Consequently, a global variable can be available to multiple namespaces, even if it is not a global percent variable. See the chapter “[Namespaces and Databases](#).”

2.4.2 The NEW Command

Caché provides another mechanism to enable you to control the scope of a variable: the **NEW** command. The argument to this command is one or more variable names, in a comma-separated list. The variables must be public variables and cannot be global variables.

This command establishes a new, limited context for the variable (which may or may not already exist). For example, consider the following routine:

ObjectScript

```
; demonew
; routine to demo NEW
NEW var2
set var1="abc"
set var2="def"
quit
```

After you run this routine, the variable `var1` is available, and the variable `var2` is not, as shown in the following example Terminal session:

```
SAMPLES>do ^demonew

SAMPLES>write var1
abc
SAMPLES>write var2

write var2
^
<UNDEFINED> *var2
```

If the variable existed before you used **NEW**, the variable still exists after the scope of **NEW** has ended, and it retains its previous value. For example, consider the following Terminal session, which uses the routine defined previously:

```
SAMPLES>set var2="hello world"

SAMPLES>do ^demonew

SAMPLES>write var2
hello world
```

2.5 Multidimensional Arrays

In ObjectScript, any variable can be a Caché *multidimensional array* (also called an *array*). A multidimensional array is generally intended to hold a set of values that are related in some way. ObjectScript provides [commands](#) and [functions](#) that provide convenient and fast access to the values; these are discussed in later sections.

You may or may not work directly with multidimensional arrays, depending on the system classes that you use and your own preferences. Caché provides a class-based alternative to use when you want a container for sets of related values; see “[Collection Classes](#),” later in this book.

2.5.1 Basics

A multidimensional array consists of any number of *nodes*, defined by subscripts. The following example sets several nodes of an array and then prints the contents of the array:

ObjectScript

```
set myarray(1)="value A"
set myarray(2)="value B"
set myarray(3)="value C"
zwrite myarray
```

This example shows a typical array. Notes:

- This array has one subscript. In this case, the subscripts are the integers 1, 2, and 3.
- There is no need to declare the structure of the array ahead of time.
- `myarray` is the name of the array itself.
- ObjectScript provides commands and functions that can act on an entire array or on specific nodes. For example:

ObjectScript

```
kill myarray
```

You can also kill a specific node and its child nodes.

- The following variation sets several subscripts of a global array named `^myglobal`; that is, these values are written to disk:

ObjectScript

```
set ^myglobal(1)="value A"  
set ^myglobal(2)="value B"  
set ^myglobal(3)="value C"
```

- There is a limit to the possible length of a global reference. This limit affects the length of the global name and the length and number of any subscripts. If you exceed the limit, you get a `<SUBSCRIPT>` error. See the section “[Maximum Length of a Global Reference](#)” in *Using Caché Globals*.
- There is a limit to the length of a value of a node. If *long strings* are not enabled in your installation, the limit is 32,767 characters. If long strings are enabled, the limit is much larger.

A later section of this book explains [how to enable long string operations](#).

A multidimensional array has one reserved memory location for each defined node and no more than that. For a global, all the disk space that it uses is dynamically allocated.

2.5.2 Structure Variations

The preceding examples show a common form of array. Note the following possible variations:

- You can have any number of subscripts. For example:

ObjectScript

```
Set myarray(1,1,1)="grandchild of value A"
```

- A subscript can be a string. The following is valid:

ObjectScript

```
set myarray("notes to self","2 Dec 2010")="hello world"
```

2.5.3 Use Notes

For those who are learning ObjectScript, a common mistake is to confuse globals and arrays. It is important to remember that any variable is either local or global, *and* may or may not have subscripts. The following table shows the possibilities:

		Example and Notes
<i>Local</i>	<i>No subscripts</i>	<pre>Set MyVar=10</pre> <p>Variables like this are quite common. The majority of the variables you see might be like this.</p>
	<i>Has subscripts</i>	<pre>Set MyVar(1)="alpha" Set MyVar(2)="beta" Set MyVar(3)="gamma"</pre> <p>A local array like this is useful when you want to pass a set of related values.</p>
<i>Global</i>	<i>No subscripts</i>	<pre>Set ^MyVar="saved note"</pre> <p>In practice, globals usually have subscripts.</p>
	<i>Has subscripts</i>	<pre>Set ^MyVar(\$USERNAME,"Preference 1")=42</pre>

2.6 Operators

This section provides an overview of the operators in ObjectScript; some are [familiar](#), and others are [not](#).

Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. You can use explicit parentheses within an expression to force certain operations to be carried out ahead of others.

Typically you use parentheses even where you do not strictly need them. It is useful to other programmers (and to yourself at a later date) to do this because it makes the intent of your code clearer.

2.6.1 Familiar Operators

ObjectScript provides the following operators for common activities:

- Mathematical operators: addition (+), subtraction (-), division (/), multiplication (*), integer division (\), modulus (#), and exponentiation (**)
- Unary operators: positive (+) and negative (-)
- String concatenation operator (_)
- Logical comparison operators: equals (=), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=)
- Logical complement operator (')

You can use this immediately before any logical value as well as immediately before a logical comparison operator.

- Operators to combine logical values: AND (&&), OR (||)

Note that ObjectScript also supports an older, less efficient form of each of these: & is a form of the && operator, and ! is a form of the || operator. You might see these older forms in existing code.

2.6.2 Unfamiliar Operators

ObjectScript also includes operators that have no equivalent in some languages. The most important ones are as follows:

- The pattern match operator (?) tests whether the characters in its left operand use the pattern in its right operand. You can specify the number of times the pattern is to occur, specify alternative patterns, specify pattern nesting, and so on.

For example, the following writes the value 1 (true) if a string (`testthis`) is formatted as a U.S. Social Security Number and otherwise writes 0.

ObjectScript

```
Set testthis="333-99-0000"  
Write testthis ?3N1 "-" 2N1 "-" 4N
```

This is a valuable tool for ensuring the validity of input data, and you can use it within the definition of class properties.

- The binary contains operator (I) returns 1 (true) or 0 (false) depending on whether the sequence of characters in the right operand is a substring of the left operand. For example:

ObjectScript

```
Set L="Steam Locomotive",S="Steam"  
Write L[I S]
```

- The binary follows operator (J) tests whether the characters in the left operand come after the characters in the right operand in ASCII collating sequence.
- The binary sorts after operator (J J) tests whether the left operand sorts after the right operand in numeric subscript collation sequence.
- The indirection operator (@) allows you to perform dynamic runtime substitution of part or all of a command argument, a variable name, a subscript list, or a pattern. Caché performs the substitution before execution of the associated command.

2.7 Commands

This section provides an overview of the commands that you are most likely to use and to see in ObjectScript. These include commands that are similar to those in other languages, as well as others that have no equivalent in other languages.

The names of commands are not case-sensitive, although they are shown in running text in all upper case by convention.

2.7.1 Familiar Commands

ObjectScript provides commands to perform familiar tasks such as the following:

- To define variables, use **SET** as shown previously.
- To remove variables, use **KILL** as shown previously.
- To control the flow of logic, use the following commands:
 - **IF**, **ELSEIF**, and **ELSE**, which work together
 - **FOR**
 - **WHILE**, which can be used on its own
 - **DO** and **WHILE**, which can be used together
 - **QUIT**, which can also return a value

There are other commands for flow control, but they are used less often.

- To trap errors, use **TRY** and **CATCH**, which work together.
- To write a value, use **WRITE**. This writes values to the current device (for example, the Terminal or a file).

Used without an argument, this command writes the values of all local variables. This is particularly convenient in the Terminal.

This command can use a small set of format control code characters that position the output. In existing code, you are likely to see the exclamation point, which starts a new line. For example:

ObjectScript

```
write "hello world",!,"another line"
```

- To read a value from the current device (for example, the Terminal), use **READ**.
 - To work with devices other than the principal device, use the following commands:
 - **OPEN** makes a device available for use.
 - **USE** specifies an open device as the current device for use by **WRITE** and **READ**.
 - **CLOSE** makes a device no longer available for use.
 - To control concurrency, use **LOCK**. Note that the Caché lock management system is different from analogous systems in other languages. It is important to review how it works; see “[Locking and Concurrency Control](#),” later in this chapter.
- You use this command in cases where multiple processes can potentially access the same variable or other item.
- To manage transactions, use **TSTART**, **TCOMMIT**, **TROLLBACK**, and related commands.
 - For debugging, use **ZBREAK** and related commands.
 - To suspend execution, use **HANG**.

2.7.2 Commands for Use with Multidimensional Arrays

In ObjectScript, you can work with multidimensional arrays in the following ways:

- To define nodes, use the **SET** command.
- To remove individual nodes or all nodes, use the **KILL** command.

For example, the following removes an entire multidimensional array:

ObjectScript

```
kill myarray
```

In contrast, the following removes the node `myarray("2 Dec 2010")` and all its children:

ObjectScript

```
kill myarray("2 Dec 2010")
```

- To delete a global or a global node but none of its descendent subnodes, use **ZKILL**.

- To iterate through all nodes of a multidimensional array and write them all, use **ZWRITE**. This is particularly convenient in the Terminal. The following sample Terminal session shows what the output looks like:

```
SAMPLES>ZWRITE ^myarray
^myarray(1)="value A"
^myarray(2)="value B"
^myarray(3)="value C"
```

This example uses a global variable rather than a local one, but remember that both can be multidimensional arrays.

- To copy a set of nodes from one multidimensional array into another, preserving existing nodes in the target if possible, use **MERGE**. For example, the following command copies an entire in-memory array (`sourcearray`) into a new global (`^mytestglobal`):

ObjectScript

```
MERGE ^mytestglobal=sourcearray
```

This can be a useful way of examining the contents of an array that you are using, while debugging your code.

2.8 Special Variables

This section introduces some Caché *special variables*. The names of these variables are not case-sensitive.

Some special variables provide information about the environment in which the code is running. These include the following:

- **\$HOROLOGY**, which contains the date and time for the current process, as given by the operating system. See “[Date and Time Values](#),” later in this chapter.
- **\$USERNAME** and **\$ROLES**, which contain information about the username currently in use, as well as the roles to which that user belongs.

ObjectScript

```
write "You are logged in as: ", $USERNAME, !, "And you belong to these roles: ", $ROLES
```

- **\$ZVERSION**, which contains a string that identifies the currently running version of Caché.

Others include **\$JOB**, **\$TIMEZONE**, **\$IO**, and **\$ZDEVICE**.

Other variables provide information about the processing state of the code. These include **\$STACK**, **\$TLEVEL**, **\$NAMESPACE**, and **\$ZERROR**.

2.8.1 \$SYSTEM Special Variable

The special variable **\$SYSTEM** provides language-independent access to a large set of utility methods.

The special variable **\$SYSTEM** is an alias for the **%SYSTEM** package, which contains classes that provide class methods that address a wide variety of needs. The customary way to refer to methods in **%SYSTEM** is to build a reference that uses the **\$SYSTEM** variable. For example, the following command executes the **SetFlags()** method in the **%SYSTEM.OBJ** class:

ObjectScript

```
DO $SYSTEM.OBJ.SetFlags("ck")
```

Because names of special variables are not case-sensitive (unlike names of classes and their members), the following commands are all equivalent:

ObjectScript

```
DO ##class(%SYSTEM.OBJ).SetFlags("ck")
DO $System.OBJ.SetFlags("ck")
DO $SYSTEM.OBJ.SetFlags("ck")
DO $system.OBJ.SetFlags("ck")
```

The classes all provide the **Help()** method, which can print a list of available methods in the class. For example:

```
SAMPLES>d $system.OBJ.Help()
'Do $system.OBJ.Help(method)' will display a full description of an individual method.

Methods of the class: %SYSTEM.OBJ

CloseObjects()
  Deprecated function, to close objects let them go out of scope.

Compile(classes,qspec,&errorlog,recurse)
  Compile a class.

CompileAll(qspec,&errorlog)
  Compile all classes within this namespace
....
```

You can also use the name of a method as an argument to **Help()**. For example:

```
SAMPLES>d $system.OBJ.Help("Compile")
Description of the method: class Compile:%SYSTEM.OBJ

Compile(classes:%String="",qspec:%String="",&errorlog:%String,recurse:%Boolean=0)
Compile a class.
<p>Compiles the class <var>classes</var>, which can be a single class, a comma separated list,
a subscripted array of class names, or include wild cards. If <var>recurse</var> is true then
do not output the initial 'compiling' message or the compile report as this is being called inside
another compile loop.<br>
<var>qspec</var> is a list of flags or qualifiers which can be displayed with
'Do $system.OBJ.ShowQualifiers()'
and 'Do $system.OBJ.ShowFlags()'
```

2.9 Locking and Concurrency Control

An important feature of any multi-process system is *concurrency control*, the ability to prevent different processes from changing a specific element of data at the same time, resulting in corruption. Consequently, ObjectScript provides a lock management system. This section provides a brief summary.

Also see “[Locks, Globals, and Namespaces](#),” later in this book.

2.9.1 Basics

The basic locking mechanism is the **LOCK** command. The purpose of this command is to delay activity in one process until another process has signaled that it is OK to proceed.

It is important to understand that a lock does not, by itself, prevent other processes from modifying the associated data; that is, Caché does not enforce unilateral locking. Locking works only by convention: it requires that mutually competing processes all implement locking with the same lock names.

You can use the **LOCK** command to create locks (replacing all previous locks owned by the process), to add locks, to remove specific locks, and to remove all locks owned by the process.

For the purpose of this simple discussion, the **LOCK** command uses the following arguments:

- The lock name. Lock names are arbitrary, but by universal convention, programmers use lock names that are identical to the names of the item to be locked. Usually the item to be locked is a global or a node of a global.
- The optional lock type (to create a non-default type of lock). There are several lock types, with different behaviors.

- An optional timeout argument, which specifies how long to wait before the attempted lock operation times out. By default, Caché waits indefinitely.

The following describes a common lock scenario: Process A issues the **LOCK** command, and Caché attempts to create a lock. If process B already has a lock with the given lock name, process A pauses. Specifically, the **LOCK** command in process A does not return, and no successive lines of code can be executed. When the process B releases the lock, the **LOCK** command in process A finally returns and execution continues.

The system automatically uses the **LOCK** command internally in many cases, such as when you work with persistent objects (discussed later in this book) or when you use certain Caché SQL commands.

2.9.2 The Lock Table

Caché maintains a system-wide, in-memory table that records all current locks and the processes that own them. This table — the lock table — is accessible via the Management Portal, where you can view the locks and (in rare cases, if needed) remove them. Note that any given process can own multiple locks, with different lock names (or even multiple locks with the same lock name).

When a process ends, the system automatically releases all locks that the process owns. Thus it is not generally necessary to remove locks via the Management Portal, except in the case of an application error.

The lock table cannot exceed a fixed size, which you can specify. For information, see “[Monitoring Locks](#)” in the *Caché Monitoring Guide*. Consequently, it is possible for the lock table to fill up, such that no further locks are possible. If this occurs, Caché writes the following message to the cconsole.log file:

```
LOCK TABLE FULL
```

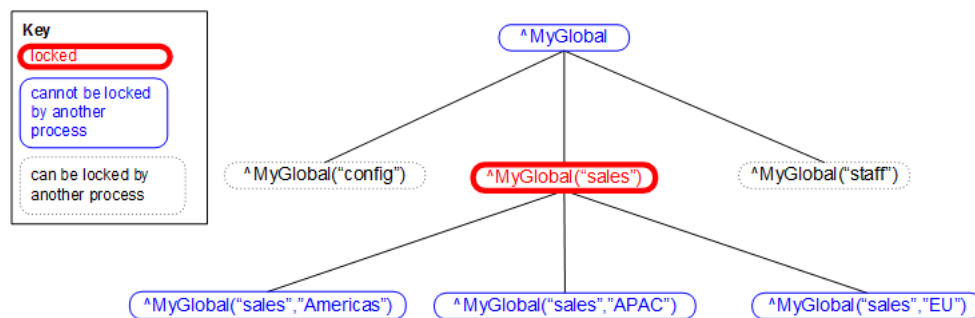
Filling the lock table is *not* generally considered to be an application error; Caché also provides a lock queue, and processes wait until there is space to add their locks to the lock table.

However, if two processes each assert an incremental lock on a variable already locked by the other process, that is a condition called *deadlock* and it *is* considered an application programming error. For details, see “[Avoiding Deadlock](#)” in the chapter “Lock Management” in *Using Caché ObjectScript*.

2.9.3 Locks and Arrays

When you lock an array, you can lock either the entire array or one or more nodes in the array. When you lock an array node, other processes are blocked from locking any node that is subordinate to that node. Other processes are also blocked from locking the direct ancestors of the locked node.

The following figure shows an example:



2.9.4 Introduction to Lock Types

When you create a lock, you specify a combination of lock type codes, which control the nature of the lock. This section discusses some of the key concepts of lock types.

Depending on the lock type, it is possible to create multiple locks with the same lock name. These locks can be owned by the same process or different processes, again depending on the lock type. The lock table displays information for all of them.

Any lock is either *exclusive* (the default) or *shared*. These types have the following significance:

- While one process has an exclusive lock (with a given lock name), no other process can acquire any lock with that lock name.
- While one process has a shared lock (with a given lock name), other processes can acquire shared locks with that lock name, but no other process can acquire an exclusive lock with that lock name.

In general, the purpose of an exclusive lock is to indicate that you intend to modify a value and that other processes should not attempt to read or modify that value. The purpose of a shared lock is to indicate that you intend to read a value and that other processes should not attempt to modify that value; they can, however, read the value.

Any lock is also either *non-escalating* (the default) or *escalating*. The purpose of escalating locks is to make it easier to manage large numbers of locks, which consume memory and which increase the chance of filling the [lock table](#). You use escalating locks when you lock multiple nodes of the same array. For escalating locks, if a given process has created more than a specific number (by default, 1000) of locks on sibling nodes of a given array, Caché removes all the individual lock names and replaces them with a new lock at the parent level. For example, you might have 1000 locks of the form `^MyGlobal("sales", "EU", salesdate)` where *salesdate* represents dates. When the same process attempts to create another lock of this form (and these locks are all escalating), Caché removes all these locks and replaces them with a lock of the name `^MyGlobal("sales", "EU")`. The lock table maintains the lock count for this new lock. This lock count is currently 1001, but when you add additional lock names of the same form, the lock table increments the lock count for the lock name `^MyGlobal("sales", "EU")`. Similarly, when you remove lock names of the same form, the lock table decrements this lock count.

There are additional subtypes of locks that Caché treats in specific ways within transactions. For details on these and for more information on locks in general, see [LOCK](#) in the *Caché ObjectScript Reference*. For information on specifying the lock threshold (which by default is 1000), see “[LockThreshold](#)” in the *Caché Parameter File Reference*.

2.10 System Functions

This section highlights some of the most commonly used system functions in ObjectScript.

The names of these functions are not case-sensitive.

The Caché class library also provides a large set of utility methods that you can use in the same way that you use functions. To find a method for a particular purpose, use the *InterSystems Programming Tools Index*.

See also “[Date and Time Values](#),” later in this chapter.

2.10.1 Value Choice

You can use the following functions to choose a value, given some input:

- **\$CASE** compares a given test expression to a set of comparison values and then returns the return value associated with the matching comparison value. For example:

```
SAMPLES>set myvar=1  
  
SAMPLES>write $CASE(myvar,0:"zero",1:"one",:"other")  
one
```

- **\$SELECT** examines a set of expressions and returns the return value associated with the first true expression. For example:

```
SAMPLES>set myvar=1  
  
SAMPLES>write $SELECT(myvar=0:"branch A",1=1:"branch B")  
branch B
```

2.10.2 Existence Functions

You can use the following functions to test for the existence of a variable or of a node of a variable.

- To test if a specific variable exists, use the **\$DATA** function.

For a variable that contains multiple nodes, this function can indicate whether a given node exists, and whether a given node has a value and child nodes.
- To get the value of a variable (if it exists) or get a default value (if not), use the **\$GET** function.

2.10.3 List Functions

ObjectScript provides a native list format. You can use the following functions to create and work with these lists:

- **\$LISTBUILD** returns a special kind of string called a *list*. Sometimes this is called *\$LIST format*, to distinguish this kind of list from other kinds (such as comma-separated lists).

The only supported way to work with a **\$LIST** list is to use the ObjectScript list functions. The internal structure of this kind of list is not documented and is subject to change without notice.
- **\$LIST** returns a list element or can be used to replace a list element.
- **\$LISTLENGTH** returns the number of elements in a list.
- **\$LISTFIND** returns the position of a given element, in a given list.

There are additional list functions as well.

If you use a list function with a value that is not a list, you receive the <LIST> error.

Note: The system class %Library.List is equivalent to a list returned by **\$LISTBUILD**. That is, when a class has a property of type %Library.List, you use the functions named here to work with that property. You can refer to this class by its short name, %List.

Caché provides other list classes that are *not* equivalent to a list returned by **\$LISTBUILD**. These are useful if you prefer to work with classes. For an introduction, see “[Collection Classes](#),” later in this book.

2.10.4 String Functions

ObjectScript also has an extensive set of functions for using strings efficiently:

- **\$EXTRACT** returns or replaces a substring, using a character count.

- **\$FIND** finds a substring by value and returns an integer specifying its end position in the string.
- **\$JUSTIFY** returns a right-justified string, padded on the left with spaces.
- **\$ZCONVERT** converts a string from one form to another. It supports both case translations (to uppercase, to lowercase, or to title case) and encoding translation (between various character encoding styles).
- **\$TRANSLATE** modifies the given string by performing a character-by-character replacement.
- **\$REPLACE** performs string-by-string replacement within a string and returns a new string.
- **\$PIECE** returns a substring from a character-delimited string (often called a *pieced string*). Many older applications use character-delimited strings as a convenient format to contain related values, each of which is a substring within the larger string. The large string acts as a record, and the substrings are its fields.

In many cases, the delimiter is a caret. Thus in existing code, you might see pieced strings like this: "value 1^value 2^value 3"

The following demonstrates how to extract a substring:

ObjectScript

```
SET mystring="value 1^value 2^value 3"
WRITE $PIECE(mystring,"^",1)
```

- **\$LENGTH** returns the number of characters in a specified string or the number of delimited substrings in a specified string, depending on the parameters used.

For example:

ObjectScript

```
SET mystring="value 1^value 2^value 3"
WRITE !, "Number of characters in this string: "
WRITE $LENGTH(mystring)
WRITE !, "Number of pieces in this string: "
WRITE $LENGTH(mystring,"^")
```

2.10.5 Working with Multidimensional Arrays

You can use the following functions to work with a multidimensional array as a whole:

- **\$ORDER** allows you to sequentially visit each node within a multidimensional array.
- **\$QUERY** enables you to visit every node and subnode within an array, moving up and down over subnodes.

To work with an individual node in an array, you can use any of the functions described previously. In particular:

- **\$DATA** can indicate whether a given node exists and whether a given node has child nodes.
- **\$GET** gets the value of a given node or gets a default value otherwise.

2.10.6 Character Values

Sometimes when you create a string, you need to include characters that cannot be typed. For these, you use **\$CHAR**.

Given an integer, **\$CHAR** returns the corresponding ASCII or Unicode character. Common uses:

- **\$CHAR(9)** is a tab.
- **\$CHAR(10)** is a line feed.
- **\$CHAR(13)** is a carriage return.

- **\$CHAR(13,10)** is a carriage return and line feed pair.

The function **\$ASCII** returns the ASCII value of the given character.

2.10.7 \$ZU Functions

In existing code, you might see items like **\$ZU(n)**, **\$ZUTIL(n)**, **\$ZU(n,n)**, and so on, where *n* is an integer. These are the *\$ZU functions*, which are now deprecated and are no longer documented. They are still available, but users are encouraged to replace them with methods and properties in the Caché class library that perform the equivalent actions. There is a table of replacements in “[Replacements for ObjectScript \\$ZUTIL Functions](#)” in the *Caché ObjectScript Reference*.

2.11 Date and Time Values

This section provides a quick overview of date and time values in ObjectScript.

2.11.1 Local Time

To access the date and time for the current process, you use the **\$HOROLOGY** special variable. Because of this, in many Caché applications, dates and times are stored and transmitted in the format used by this variable. This format is often called *\$H format* or *\$HOROLOGY format*.

\$HOROLOGY retrieves the date and time from the operating system and is thus always in the local time zone.

The Caché class library includes data type classes to represent dates in more common formats such as ODBC, and many applications use these instead of \$H format.

2.11.2 UTC Time

Caché also provides the **\$ZTIMESTAMP** special variable, which contains the current date and time as a Coordinated Universal Time value in \$H format. This is a worldwide time and date standard; this value is very likely to differ from your local time (and date) value.

2.11.3 Date and Time Conversions

ObjectScript includes functions for converting date and time values.

- Given a date in \$H format, the function **\$ZDATE** returns a string that represents the date in your specified format.

For example:

```
SAMPLES>WRITE $ZDATE($HOROLOGY,3)
2010-12-03
```

- Given a date and time in \$H format, the function **\$ZDATETIME** returns a string that represents the date and time in your specified format.

For example:

```
SAMPLES>WRITE $ZDATETIME($HOROLOGY,3)
2010-12-03 14:55:48
```

- Given string dates and times in other formats, the functions **\$ZDATEH** and **\$ZDATETIMEH** convert those to \$H format.

- The functions **\$ZTIME** and **\$ZTIMEH** convert times from and to \$H format.

2.11.4 Details of the \$H Format

The \$H format is a pair of numbers separated by a comma. For example: 54321,12345

- The first number is the number of days since December 31st, 1840. That is, day number 1 is January 1st, 1841. This number is always an integer.
- The second number is the number of seconds since midnight on the given day.
Some functions, such as **\$NOW()**, provide a fractional part.

For additional details, including an explanation of the starting date, see **\$SHOROLOG** in the *Caché ObjectScript Reference*.

2.12 Using Macros and Include Files

As noted earlier, you can define macros and use them later in the same routine. More commonly, you define them in include files. An include file is a document with the extension .inc in Studio.

To define a macro, use the **#define** directive or other preprocessor directive. For example:

ObjectScript

```
#define mymacro "hello world"
```

To include an include file in a routine, use the **#include** directive. For example:

ObjectScript

```
#include myincludefile
```

(Note that the **syntax** is slightly different in class definitions.)

To refer to a macro, use the following syntax:

```
$$$macroname
```

Or:

```
$$$macroname(arguments)
```

The preprocessor directives are documented in “[ObjectScript Macros and the Macro Preprocessor](#)” in Using Caché ObjectScript.

Note: Both Studio and the Management Portal list the include files with the routines. For example, the Studio Workspace window shows include files within the **Routines** folder. Include files are not, however, actually routines because they are not executable.

2.13 Using Routines and Subroutines

To execute a routine, you use the DO command, as follows:

ObjectScript

```
do ^routinename
```

To execute a procedure, function, or subroutine (without accessing its return value), you use the following command:

ObjectScript

```
do label^routinename
```

Or:

ObjectScript

```
do label^routinename(arguments)
```

To execute a procedure, function, or subroutine and refer to its return value, you use an expression of the form `$$label^routinename` or `$$label^routinename(arguments)`. For example:

ObjectScript

```
set myvariable=$$label^routinename(arguments)
```

In all cases, if the label is within the same routine, you can omit the caret and routine name. For example:

ObjectScript

```
do label
do label(arguments)
set myvariable=$$label(arguments)
```

In all cases, the arguments that you pass can be either literal values, expressions, or names of variables.

2.13.1 Passing Variables by Value or by Reference

When you invoke code, you can pass values of variables to that code either by value or by reference. In most cases, these variables are local variables with no subscripts, so this section discusses those first.

As with other programming languages, Caché has a memory location that contains the value of each local variable. The name of the variable acts as the address to the memory location.

When you pass a local variable with no subscripts to procedure, function, or subroutine, you pass the variable *by value*. This means that the system makes a copy of the value, so that the original value is not affected. To pass the memory address instead, place a period immediately before the name of the variable in the argument list. For example:

ObjectScript

```
do ^myroutine(.myarg)
```

To demonstrate this, consider the following procedure:

ObjectScript

```
square(input) public
{
    set answer=input*input
    set input=input + 10
    quit answer
}
```


Suppose that you define a variable and pass it by value to this procedure:

```
SAMPLES>set myvariable=5
SAMPLES>write $$square^demobyref(myvariable)
25
SAMPLES>write myvariable
5
```

In contrast, suppose that you pass the variable by reference:

```
SAMPLES>set myvariable=5
SAMPLES>write $$square^demobyref(.myvariable)
25
SAMPLES>write myvariable
15
```

There are other variations of variables in addition to local variables with no subscripts. The following table summarizes all the possibilities:

Kind of Variable	Passing by Value	Passing by Reference
Local variable with no subscripts	The default way in which these variables are passed	Allowed
Local variable (with subscripts)	Cannot be passed this way (only the top node would be passed)	Required
Global variable (with or without subscripts)	Required	Cannot be passed this way (data for a global is not in memory)

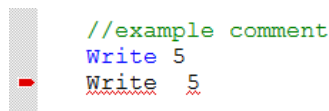
2.14 Potential Pitfalls

The following items can confuse programmers who are new to ObjectScript, particularly if those who are responsible for maintaining code written by other programmers:

- Within a routine or a method, every line must be indented by at least one space or one tab unless that line contains a label. That is, if there is text of any kind in the first character position, the compiler and Studio treat it as a label. Note that Studio displays labels in red, as seen in previous examples.

There is one exception: A curly brace is accepted in the first character position.

- There must be exactly one space (not a tab) between a command and its first argument. Otherwise, Studio indicates that you have a syntax error:



```
//example comment
Write 5
Write 5
```

Similarly, the Terminal displays a syntax error as follows:

```
SAMPLES>write 5
WRITE 5
^
<SYNTAX>
SAMPLES>
```

- Operator precedence in ObjectScript is strictly left-to-right; within an expression, operations are performed in the order in which they appear. You can use explicit parentheses within an expression to force certain operations to be carried ahead of others.

Typically you use parentheses even where you do not strictly need them. It is useful to other programmers (and to yourself at a later date) to do this because it makes the intent of your code clearer.

- For reasons of history, ObjectScript does not consider an empty string (" ") to equal the ASCII NULL value. To represent the ASCII NULL value, use `$CHAR(0)`. (**\$CHAR** is a system function that returns an ASCII character, given its decimal-based code.) For example:

ObjectScript

```
write "" = $char(0)
```

Similarly, when ObjectScript values are projected to SQL or XML, the values " " and `$CHAR(0)` are treated differently. For information on the SQL projections of these values, see “[Null and the Empty String](#)” in the chapter “Language Elements” in *Using Caché SQL*. For information on the XML projections of these values, see the chapter “[Handling Empty Strings and Null Values](#)” in *Projecting Objects to XML*.

- Some parts of ObjectScript are case-sensitive while others are not. The case-insensitive items include names of commands, functions, special variables, namespaces, and users.

The case-sensitive items include names of most of the elements that you define: routines, variables, classes, properties, and methods. For more details, see “[Syntax Rules](#)” in *Using Caché ObjectScript*.

- Most command names can be represented by an abbreviated form. Therefore, **WRITE**, **Write**, **write**, **W**, and **w** are all valid forms of the **WRITE** command. For a list, see “[Table of Abbreviations](#)” in the *Caché ObjectScript Reference*.
- For many of the commands, you can include a *postconditional expression* (often simply called a *postconditional*).

This expression controls whether Caché executes the command. If the postconditional expression evaluates to true (nonzero), Caché executes the command. If the expression evaluates to false (zero), Caché ignores the command and continues with the next command.

For example:

ObjectScript

```
Set count = 6
Write:count<5 "Print this if count is less than five"
Write:count>5 "Print this if count is greater than five"
```

The preceding generates the following output: `Print this if count is greater than five`

Note: If postconditionals are new to you, you might find the phrase “postconditional expression” somewhat misleading, because it suggests (incorrectly) that the expression is executed *after* the command. Despite the name, a postconditional is executed *before* the command.

- You can include multiple commands on a single line. For example:

ObjectScript

```
set myval="hello world" write myval
```

When you do this, beware that you must use two spaces after any command that does not take arguments, if there are additional commands on that line; if you do not do so, a syntax error occurs.

- The **IF**, **ELSE**, **FOR**, and **DO** commands are available in two forms:
 - A newer block form, which uses curly braces to indicate the block. For example:

ObjectScript

```
if (testvalue=1) {
    write "hello world"
}
```

InterSystems recommends that you use the block form in all new code.

- An older line-based form, which does not use curly braces. For example:

ObjectScript

```
if (testvalue=1) write "hello world"
```

- As a result of the preceding items, ObjectScript can be written in a very compact form. For example:

ObjectScript

```
s:$g(%d(3))'=" " %d(3)=$$fdN3(%d(3)) q
```

The class compiler automatically generates compact code of the form shown above (although not necessarily with abbreviated commands as in this example). Sometimes it is useful to look at this generated code, to track down the source of a problem or to understand how something works.

Many older applications are written in the compact form shown here.

- There are no truly reserved words in ObjectScript, so it is theoretically possible to have a variable named `set`, for example. However, it is prudent to avoid names of commands, functions, SQL reserved words, and certain system items; see “[Syntax Rules](#)” in *Using Caché ObjectScript*.
- Caché allocates a fixed amount of memory to hold the results of string operations. If a string expression exceeds the amount of space allocated, a <MAXSTRING> error results. If long strings are not enabled, the limit is 32,767 characters. If long strings are enabled, the limit is much larger. A later section of this book explains how to enable [long string operations](#).

For class definitions, the string operation limit affects the size of string properties. Caché provides a system object (called a *stream*) that you can use when you need to work with strings that exceed this limit. A later section of this book explains how to use the [stream interface classes](#).

2.15 For More Information

The chapters after this provide more detail on the topics covered in this chapter. This information is taken from the following books:

- [Using Caché ObjectScript](#) provides details on ObjectScript.
- [Using Caché Globals](#) provides details on multidimensional arrays and globals.
- [Caché ObjectScript Reference](#) provides reference information for the operators, commands, functions, special variables, and other parts of ObjectScript.

The Caché documentation also includes books on Caché MVBasic and Caché Basic, which this book does not discuss in detail.

3

Basic Ideas in Class Programming

If you are not familiar with class programming, this chapter is intended to give you a sense of how this kind of programming works. If you are familiar with class programming, you might find it helpful just to skim the code examples, so that you see what class programming in Caché looks like.

This chapter discusses the following:

- [Objects and properties](#)
- [Methods](#)
- [Class constants \(parameters\)](#)
- [Class definitions and the use of types](#)
- [Inheritance](#)
- [Classes as containers of methods](#)
- [Abstract classes](#)

The concepts in this chapter are largely independent of language, although the examples use ObjectScript.

3.1 Objects and Properties

In class programming, a key concept is *objects*. An object is a container for a set of values that are stored together or passed together as a set. An object often corresponds to a real-life entity, such as a patient, a patient diagnosis, a transaction, and so on.

A class definition is often a template for objects of a given type. The class definition has properties to contain the values for those objects. For example, suppose that we have a class named MyApp.Clinical.PatDiagnosis; this class could have the properties Date, EnteredBy, PatientID, DiagnosedBy, Code, and others.

You use the template by creating *instances* of the class; these instances are objects. For example, suppose that a user enters a patient diagnosis into a user interface and saves that data. The underlying code would have the following logic:

1. Create a new patient diagnosis object from the patient diagnosis template.
2. Set values for the properties of the object, as needed. Some may be required, some may have default values, some may be calculated based on others, and some may be purely optional.
3. Save the object.

This action stores the data.

The following shows an example that uses ObjectScript:

ObjectScript

```
//create the object
set diagnosis=##class(MyApp.Clinical.PatDiagnosis).%New()

//set a couple of properties by using special variables
set diagnosis.Date=$SYSTEM.SYS.TimeStamp()
set diagnosis.EnteredBy=$username

//set other properties based on variables set earlier by
//the user interface
set diagnosis.PatientID=patientid
set diagnosis.DiagnosedBy=clinicianid
set diagnosis.Code=diagcode

//save the data
//the next line tries to save the data and returns a status to indicate
//whether the action was successful
set status=diagnosis.%Save()
//always check the returned status
if $$$ISERR(status) {do $System.Status.DisplayError(status) quit status}
```

Note the following points:

- To refer to a property of the object, you use the syntax *object_variable.property_name*, for example:
diagnosis.DiagnosedBy
- **%New()** and **%Save()** are methods of the MyApp.Clinical.PatDiagnosis class.

The next section discusses types of methods and why you invoke them in different ways as seen here.

3.2 Methods

A method is a procedure (in most cases; Caché supports other kinds of methods as you will see in the next chapter). Methods can invoke each other and can refer to properties and parameters. The next chapter gives the [details for the rules in Caché](#).

There are two kinds of methods in a class language: instance methods and class methods. These have different purposes and are used in different ways.

3.2.1 Instance Methods

An *instance method* has meaning only when invoked from an instance of the class, usually because you are doing something to or something with that instance. For example:

ObjectScript

```
set status=diagnosis.%Save()
```

For example, suppose that we are defining a class that represents patients. In this class, we could define instance methods to perform the following actions:

- Calculate the BMI (body mass index) for the patient
- Print a report summarizing information for the patient
- Determine whether the patient is eligible for a specific procedure

Each of these actions requires knowledge of data stored for the patient, which is why most programmers would write them as instance methods. Internally, the implementation of an instance method typically refers to properties of that instance. The following shows an example definition of an instance method that refers to two properties:

Class Member

```
Method GetBMI() as %Numeric
{
  Set bmi=..WeightKg / (..HeightMeter**2)
  Quit bmi
}
```

To use this method, your application code might include lines like this:

ObjectScript

```
//open the requested patient given an id selected earlier
set patient=##class(MyApp.Clinical.PatDiagnosis).%OpenId(id)

//get value to display in BMI Display field
set BMIDisplay=patient.GetBMI()
```

3.2.2 Class Methods

The other type of method is the *class method* (called a *static method* in other languages). To invoke this type of method, you use syntax that does not refer to an instance. For example:

ObjectScript

```
set patient=##class(MyApp.Clinical.PatDiagnosis).%New()
```

There are three very general reasons to write class methods:

- You need to perform an action that creates an instance of the class.

By definition, this action cannot be an instance method.

- You need to perform an action that affects multiple instances.

For example, you might need to reassign a group of patients to a different primary care physician.

- You need to perform an action that does not affect any instance.

For example, you can write a method that returns the time of day, or a random number, or a string formatted in a particular way.

3.2.3 Methods and Variable Scope

A method typically sets values of variables. In nearly all cases, these variables are available only within this method. For example, consider the following class:

Class Definition

```
Class GORIENT.VariableScopeDemo
{
  ClassMethod Add(arg1 As %Numeric, arg2 As %Numeric) As %Numeric
  {
    Set ans=arg1+arg2
    Quit ans
  }

  ClassMethod Demo1()
  {
    set x=..Add(1,2)
  }
}
```

```
    write x
}

ClassMethod Demo2()
{
    set x=..Add(2,4)
    write x
}
}
```

The **Add()** method sets a variable named `ans` and then returns the value contained in that variable.

The method **Demo1()** invokes the method **Add()**, with the arguments 1 and 2, and then writes the answer. The method **Demo2()** is similar but uses different hardcoded arguments.

If the method **Demo1()** or **Demo2()** tried to refer to the variable `ans`, that variable would be undefined in that context and Caché would throw an error.

Similarly, **Add()** cannot refer to the variable `x`. Also the variable `x` within **Demo1()** is a different variable from the variable `x` within **Demo2()**.

These variables have limited scope because that is the default behavior of Caché classes (and is the usual behavior in other class languages).

Within class definitions, you pass values almost entirely by including them as arguments to methods. This is the convention in class programming. This convention simplifies the job of determining the scope of variables.

In contrast, when you write routines, it is necessary to understand the rules that control scoping. These are discussed in [Using Caché ObjectScript](#).

3.3 Class Constants (Parameters)

Sometimes it is useful for a class to have easy access to a constant value. In Caché classes, such a value is a *class parameter*. Other languages use the term *class constant* instead. The following shows an example:

Class Member

```
Parameter MYPARAMETER = "ABC" ;
```

A class parameter acquires a value at compile time and cannot be changed later.

Your methods can refer to parameters; that is why you define parameters. For example:

ObjectScript

```
set myval=..#MYPARAMETER * inputvalue
```

3.4 Class Definitions and Types

The following shows an example of a class definition, which we will use to discuss types in class definitions:

Class Definition

```

Class MyClass Extends %Library.Persistent
{
  Parameter MYPARAMETER = "ABC" ;

  Property DateOfBirth As %Library.Date;
  Property Home As Sample.Address;

  Method CurrentAge() As %Library.Integer
  {
    //details
  }

  ClassMethod Addition(x As %Library.Integer, y As %Library.Integer) As %Library.Integer
  {
    //details
  }
}

```

This class definition defines one parameter (MYPARAMETER), two properties (DateOfBirth and Home), one instance method (CurrentAge()), and one class method (Addition()).

In class programming, you can specify *types* in the following key places:

- For the class itself. The element after `Extends` is a type.
Each type is the name of class.
- For parameters. In this case and in the remaining cases, the element after `As` is a type.
- For properties. For the `Home` property, the type is a class that itself contains properties.
In this case, the type has an *object value*. In the example here, this is an *object-valued property*.
Object-valued properties can contain other object-valued properties.
- For the return value of a method.
- For the value of any arguments used by a method.

3.5 Inheritance

In most class-based languages, a major feature is *inheritance*: one class can inherit from other classes and thus acquire the parameters, properties, methods, and other elements of those other classes. Collectively, the parameters, properties, methods, and other elements are known as *class members*.

3.5.1 Terminology and Basics

When class A inherits from class B, we use the following terminology:

- Class A is a *subclass* of class B. Alternatively, class A *extends* class B.
Sometimes it is said that class A is a *subtype* of class B.
- Class B is a *superclass* of class A.
Sometimes it is said that class A is the *child class* and class B is the *parent class*. This terminology is common but can be misleading, because the words *parent* and *child* are used in quite a different sense when discussing SQL tables.

When a class inherits from other classes, it acquires the class members of those other classes, including members that the superclasses have themselves inherited. The subclass can override the inherited class members.

It is possible for multiple superclasses of one class to define methods with the same name, properties with the same name, and so on. Therefore it is necessary to have rules for deciding which superclass contributes the definition that is used in the subclass. (See “[Inheritance Rules in Caché](#),” in the next chapter.)

In the Caché class library, superclasses usually have different purposes and have members with different names, and conflicts of member names are not common.

3.5.2 Example

The following shows an example from Ensemble:

```
/// Finds files in a FilePath directory and submits all that match a FileSpec wildcard to
/// an associated BusinessService for processing within Ensemble
Class EnsLib.File.InboundAdapter Extends (Ens.InboundAdapter, EnsLib.File.Common)
```

This example is presented solely to demonstrate how a class can combine logic from different superclasses. This `EnsLib.File.InboundAdapter` class inherits from two classes that do quite different things:

- `Ens.InboundAdapter`, which contains the basic logic for something known as an “inbound adapter,” a concept in Ensemble.
- `EnsLib.File.Common`, which contains logic for working with sets of files in a given directory.

In `EnsLib.File.InboundAdapter`, the methods use logic from both of these classes, as well as their superclasses.

3.5.3 Use of Inherited Class Members

When you see the definition of a class in an editing tool, you do not see the inherited members that it contains but your code can refer to them.

For example, suppose that class A has two properties, each of which has a default value, as follows:

Class Definition

```
Class Demo.A
{
  Property Prop1 as %Library.String [InitialExpression = "ABC"];
  Property Prop2 as %Library.String [InitialExpression = "DEF"];
}
```

Class B could look like this:

Class Definition

```
Class Demo.B Extends Demo.A
{
  Method PrintIt()
  {
    Write ..Prop1,!
    Write ..Prop2,!
  }
}
```

As noted earlier, a subclass can override the inherited class members. For example, class C could also inherit from class A but could override the default value of one of its properties:

Class Definition

```
Class Demo.C Extends Demo.A
{
Property Prop2 as %Library.String [InitialExpression = "GHI"];
}
```

3.5.4 Use of Subclasses

If class B inherits from class A, you can use an instance of class B in any location where you can use an instance of class A.

For example, suppose that you have a utility method like the following:

Class Member

```
ClassMethod PersonReport(person as MyApp.Person) {
//print a report that uses properties of the instance
}
```

You can use an instance of `MyApp.Person` as input to this method. You can also use an instance of any subclass of `MyApp.Person`. For example:

ObjectScript

```
//id variable is set earlier in this program
set employee=##class(MyApp.Employee).%OpenId(id)
do ##class(Util.Utls).PersonReport(employee)
```

Similarly, the return value of a method (if it returns a value) can be an instance of a subclass of the specified type. For example, suppose that `MyApp.Employee` and `MyApp.Patient` are both subclasses of `MyApp.Person`. You could define a method as follows:

Class Member

```
ClassMethod ReturnRandomPerson() as MyApp.Person
{
Set randomnumber = $RANDOM(10)
If randomnumber > 5 {
set person=##class(MyApp.Employee).%New()
}
else {
set person=##class(MyApp.Patient).%New()
}
quit person
}
```

3.6 Classes as Containers of Methods

As noted earlier, a class definition is often a template for objects. Another possibility is for a class to be a container for a set of class methods that belong together. In this case, you never create an instance of this class. You only invoke class methods in it.

For examples, see the classes in the Caché %SYSTEM package, which is introduced earlier in the section “[\\$SYSTEM Special Variable](#).”

3.7 Abstract Classes

It is also useful to define abstract classes. An *abstract class* typically defines a generic interface and cannot be instantiated. A method definition within the class declares the signature of the method, but not its implementation.

You define an abstract class to describe an interface. Then you or other developers create subclasses, and in those subclasses, implement the methods. The implementation must match the signature specified in the abstract class. This system enables you to develop multiple, parallel classes with slightly different purposes but identical interfaces. Many of the system classes have common interfaces for this reason; for example, see the stream classes and collection classes introduced in the chapter “[Caché Objects](#).”

It is also possible to specify that a method is abstract even if the class is not.

4

Caché Classes

It is useful to review the basic rules for working with and creating classes in general, before looking at the object classes in particular. Thus this chapter discusses the basic rules for defining and working with classes in Caché. It discusses the following topics:

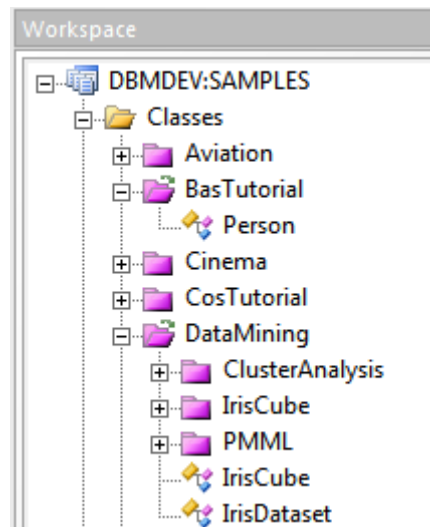
- [Class names and packages](#)
- [Basic contents of a class definition](#)
- [Shortcuts for class names](#)
- [Parameters](#)
- [Properties](#)
- [Properties based on data types](#)
- [Methods](#)
- [Method arguments](#)
- [Special kinds of methods](#)
- [Class queries](#)
- [XData blocks](#)
- [Macros and include files](#)
- [Inheritance rules in Caché](#)
- [Other sources of information on these topics](#)

The [next chapter](#) discusses objects and object classes. Namespaces are discussed [later](#) in this book.

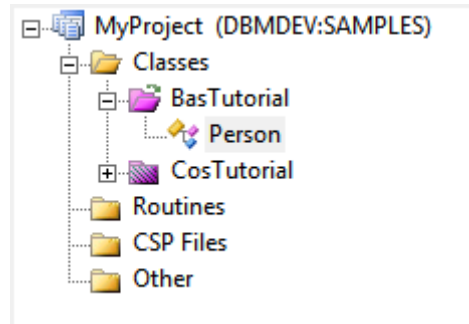
4.1 Class Names and Packages

Each Caché class has a name, which must be unique within the [namespace](#) where it is defined. A *full class name* is a string delimited by one or more periods, as in the following example: `package.subpackage.subpackage.class`. The *short class name* is the part after the final period within this string; the part preceding the final period is the *package name*.

The package name is simply a string, but if it contains periods, the Caché development tools treat each period-delimited piece as a *subpackage*. Studio and other tools display these subpackages as a hierarchy of folders, for convenience. For example, the **Namespace** tab of the Studio Workspace window displays packages like this:



Note that **Project** tab of the Studio Workspace window shows the contents of the currently selected Studio project. In this tab, if the project includes the entire package or subpackage, the package icon is a folder with slanted blue lines. If only some classes in the package are in the project, Studio uses the usual icon. The following shows an example:



This book does not discuss projects in any detail; for information, see [Using Studio](#).

Studio provides options for exporting individual classes, entire packages, and projects to XML files, as well as for importing those items from XML files.

4.2 Basic Contents of a Class Definition

A Caché class definition can include the following items, all known as *class members*:

- **Parameters** — A parameter defines a constant value for use by this class. The value is set at compilation time.
- **Methods** — There are two kinds of methods: instance methods and class methods (called static methods in other languages). In most cases, a method is a subroutine.
- **Properties** — A property contains data for an instance of the class.
- **Class queries** — A class query defines an SQL query that can be used by the class and specifies a class to use as a container for the query.
- **XData blocks** — An XData block is a well-formed XML document within the class, for use by the class.

These have many possible applications.

- Other kinds of class members that are relevant only for persistent classes; these are discussed in the [next chapter](#).

A class definition can include *keywords*; these affect the behavior of the class compiler. You can specify some keywords for the entire class, and others for specific class members. These keywords affect the code that the class compiler generates and thus control the behavior of the class.

The following shows a simple Caché class definition:

Class Definition

```
Class MyApp.Main.SampleClass Extends %RegisteredObject
{
    Parameter CONSTANTMESSAGE [Internal] = "Hello world!" ;

    Property VariableMessage As %String [ InitialExpression = "How are you?"];

    Property MessageCount As %Numeric [Required];

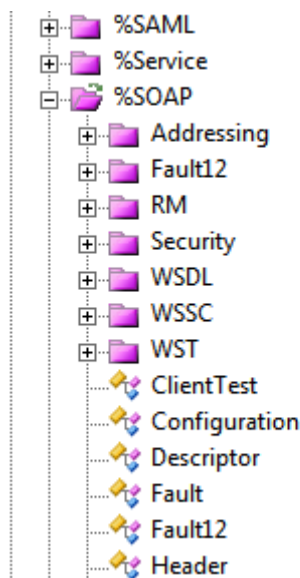
    ClassMethod HelloWorld() As %String
    {
        Set x=..#CONSTANTMESSAGE
        Quit x
    }

    Method WriteIt() [ ServerOnly = 1]
    {
        Set count=..MessageCount
        For i=1:1:count {
            Write !,..#CONSTANTMESSAGE," " ,..VariableMessage
        }
    }
}
```

Note the following points:

- The first line gives the name of the class. `MyApp.Main.SampleClass` is the full class name, `MyApp.Main` is the package name, and `SampleClass` is the short class name.

Studio and other user interfaces treat each package as a folder. For example:



- You cannot edit the class name. (If you do, your changes are ignored, and the original class name is shown when you next open the class definition.) For information on renaming a class, see “[Renaming a Class](#),” later in this book.
- `Extends` is a compiler keyword.

The `Extends` keyword specifies that this class is a superclass of `%RegisteredObject`, which is a system class, discussed in the next chapter. This class extends only one class, but it is possible to extend multiple other classes. Those classes, in turn, can extend other classes.

- `CONSTANTMESSAGE` is a parameter. By convention, all parameters in Caché system classes have names in all capitals. This is a convenient convention, but you are not required to follow it.

The `Internal` keyword is a compiler keyword. It marks this parameter as internal, which suppresses it from display in the class documentation. This parameter has a string value.

- `VariableMessage` and `MessageCount` are properties. The item after `As` indicates the types for these properties. `InitialExpression` and `Required` are compiler keywords.
- `HelloWorld()` is a class method and it returns a string; this is indicated by the item after `As`.

This method uses the value of the class parameter.

- `WriteIt()` is an instance method and it does not return a value.

This method uses the value of the class parameter and values of two properties.

The `ServerOnly` compiler keyword means that this method will not be projected to Java or C++ clients.

The following Terminal session shows how we can use this class:

```
SAMPLES>write ##class(MyApp.Main.SampleClass).HelloWorld()  
Hello world!  
SAMPLES>set x=##class(MyApp.Main.SampleClass).%New()  
  
SAMPLES>set x.MessageCount=3  
  
SAMPLES>do x.WriteIt()  
  
Hello world! How are you?  
Hello world! How are you?  
Hello world! How are you?
```

4.3 Class Name Shortcuts

When referring to a class, you can omit the package (or the higher level packages) in the following scenarios:

- The reference is within a class, and the referenced class is in the same package or subpackage.
- The reference is within a class, and the class uses the `IMPORT` directive to import the package or subpackage that contains the referenced class.
- The reference is within a method, and the method uses the `IMPORT` directive to import the package or subpackage that contains the referenced class.
- You are referring to a class in the `%Library` package, which is specially handled. You can refer to the class `%Library.ClassName` as `%ClassName`. For example, you can refer to `%Library.String` as `%String`.
- You are referring to a class in the `User` package, which is specially handled. For example, you can refer to `User.MyClass` as `MyClass`.

`InterSystems` does not provide any classes in the `User` package, which is reserved for your use.

4.4 Class Parameters

A class parameter defines a value that is the same for all objects of a given class. This value is established when the class is compiled and cannot be altered at runtime. You use class parameters for the following purposes:

- To define a value that should not be changed at runtime.
- To define user-specific information about a class definition. A class parameter is simply an arbitrary name-value pair; you can use it to store any information you like about a class.
- To customize the behavior of the various data type classes (such as providing validation information) when used as properties; this is discussed in the next section.
- To provide parameterized values for [method generator](#) methods to use.

The following shows a class with several parameters:

Class Definition

```
Class GSOP.DivideWS Extends %SOAP.WebService
{
    Parameter USECLASSNAMESPACES = 1;

    /// Name of the Web service.
    Parameter SERVICENAME = "Divide";

    /// SOAP namespace for the Web service
    Parameter NAMESPACE = "http://www.mynamespace.org";

    /// let this Web service understand only SOAP 1.2
    Parameter SOAPVERSION = "1.2";

    ///further details omitted
}
```

4.5 Properties

Formally, there are two kinds of properties in Caché:

- Attributes, which hold values. The value can be any of the following:
 - A single, literal value, usually based on a data type.
 - An object value (this includes collection objects and stream objects, both introduced in the next chapter).
 - A multidimensional array. This is less common.

The word *property* often refers just to properties that are attributes, rather than properties that hold associations.

- Relationships, which hold associations between objects.

This section shows a sample class that contains property definitions that show some of these variations:

Class Definition

```
Class MyApp.Main.Patient Extends %Persistent
{
    Property PatientID As %String [Required];
    Property Gender As %String(DISPLAYLIST = ",Female,Male", VALUELIST = ",F,M");
    Property BirthDate As %Date;
    Property Age As %Numeric [Transient];
    Property MyTempArray [MultiDimensional];
    Property PrimaryCarePhysician As Doctor;
    Property Allergies As list Of PatientAllergy;
    Relationship Diagnoses As PatientDiagnosis [ Cardinality = children, Inverse = Patient ];
}
```

Note the following:

- In each definition, the item after `As` is the type of the property. Each type is a class. The syntax `As List Of` is shorthand for a specific collection class; these are discussed in the next chapter.

 `%String`, `%Date`, and `%Numeric` are data type classes.

 `%String` is the default type.
- `Diagnoses` is a relationship property; the rest are attribute properties.
- `PatientID`, `Gender`, `BirthDate`, and `Age` can contain only simple, literal values.
- `PatientID` is required because it uses the `Required` keyword. This means that you cannot save an object of this class if you do not specify a value for this property.
- `Age` is not saved to disk, unlike the other literal properties. This is because it uses the `Transient` keyword.
- `MyTempArray` is a *multidimensional property* because it uses the `MultiDimensional` keyword. This property is not saved to disk by default.
- `PrimaryCarePhysician` and `Allergies` are *object-valued properties*.
- The `Gender` property definition includes values for property parameters. These are parameters in the data type class that this property uses.

This property is restricted to the values `M` and `F`. When you display the logical values (as in the Management Portal), you see `Male` and `Female` instead. Each data type class provides methods such as **`LogicalToDisplay()`**.

Object-valued properties and relationship properties are discussed in the next chapter.

4.5.1 Specifying Property Keywords

In a property definition, you can include optional property keywords that affect how the property is used. The following list shows some of the most commonly seen keywords:

Required

Specifies that the value of the property set before an instance of this class can be stored to disk. By default, properties are not required. In a subclass, you can mark an optional property as required, but you cannot do the reverse.

InitialExpression

Specifies an initial value for the property. By default, properties have no initial value. Subclasses inherit the value of the InitialExpression keyword and can override it. The value specified must be a valid ObjectScript expression (this applies even if the class is written in another language, such as Caché MVBasic).

Transient

Specifies that the property is not stored in the database. By default, properties are not transient. Subclasses inherit the value of the Transient keyword and cannot override it.

Private

Specifies that the property is private. Subclasses inherit the value of the Private keyword and cannot override it.

By default, properties are public and can be accessed anywhere. You can mark a property as private (via the Private keyword). If so, it can only be accessed by methods of the object to which it belongs.

In Caché, private properties are always inherited and visible to subclasses of the class that defines the property.

In other programming languages, these are often called *protected properties*.

Calculated

Specifies that the property has no in-memory storage allocated for it when the object containing it is instantiated. By default, a property is not calculated. Subclasses inherit the Calculated keyword and cannot override it.

MultiDimensional

Specifies that the property is multidimensional. This property is different from other properties as follows:

- It does not have associated methods (see the following topics).
- It is ignored when the object is validated or saved.
- It is not saved to disk, unless your application includes code to save it specifically.
- It cannot be exposed through ActiveX or Java.
- It cannot be stored in or exposed through SQL tables.

Multidimensional properties are rare but are occasionally useful to temporarily contain object state information.

4.6 Properties Based on Data Types

When you define a property and you specify its type as a data type class, you have special options for defining and working with that property, as described in this section.

4.6.1 Data Type Classes

Data type classes enable you to enforce sets of rules about the values of properties.

Caché provides data type classes which include %Library.String, %Library.Integer, %Library.Numeric, %Library.Date, %MV.Date, and many others. Because the names of classes of the %Library package can be abbreviated, you can abbreviate many of these; for example, %Date is an abbreviation for %Library.Date.

Each data type class has the following features:

- It specifies values for compiler keywords. For a property, a compiler keyword can do things like the following:
 - Make the property required
 - Specify an initial value for the property
 - Control how the property is projected to SQL, ODBC, ActiveX, and Java clients
- It specifies values for parameters that affect the details such as the following:
 - Maximum and minimum allowed logical value for the data type
 - Maximum and minimum number of characters the string can contain
 - Number of digits following the decimal point
 - Whether to truncate the string if it exceeds the maximum number of characters
 - Display format
 - How to escape any special XML or HTML characters
 - Enumerated lists of logical values and display values to use in any user interface
 - Pattern that the string must match (automatically uses the Caché pattern-matching operator)
 - Whether to respect or ignore the UTC time zone when importing or exporting to XML
- It provides a set of methods to translate literal data among the stored (on disk), logical (in memory), and display formats.

You can add your own data type classes. For example, the following shows a custom subclass of **%Char**:

Class Definition

```
Class MyApp.MyType Extends %Library.Char
{
  /// The maximum number of characters the string can contain.
  Parameter MAXLEN As INTEGER = 2000;
}
```

4.6.2 Overriding Parameters of Data Type Classes

When you define a property and you specify its type as a data type class, you can override any parameters defined by the data type class.

For example, the **%Integer** data type class defines the class parameter (*MAXVAL*) but provides no value for this parameter. You can override this in a property definition as follows:

Class Member

```
Property MyInteger As %Integer(MAXVAL=10);
```

For this property, the maximum allowed value is 10.

(Internally, this works because the validation methods for the data type classes are method generators; the parameter value you provide is used when the compiler generates code for your class. Method generators are discussed later in “[Special Kinds of Methods](#).”)

Similarly, every property of type **%String** has a collation type, which determines how values are ordered (such as whether capitalization has effects or not). The default collation type is **SQLUPPER**. For more details on collations, see the section “[Data Collation](#)” in the chapter “Caché SQL Basics” in *Using Caché SQL*.

For another example, the data type classes define the *DISPLAYLIST* and *VALUELIST* parameters, which you can use to specify choices to display in a user interface and their corresponding internal values:

```
Property Gender As %String(DISPLAYLIST = ",Female,Male", VALUELIST = ",F,M");
```

4.6.3 Using Instance Variables

To access the in-memory value of a property from within an instance method of an object, you can use the following in-memory value syntax:

ObjectScript

```
Set i%Name = "Carl"
```

This directly sets “Carl” as the in-memory value of the property Name, bypassing the **NameSet** *accessor method* (if present). The variable *i%Name* is an *instance variable*; see “*i%<PropertyName> syntax*” in *Using Caché Objects*. For information on accessor methods, see the chapter “*Using and Overriding Property Methods*” in the same book.

4.6.4 Using Other Property Methods

Properties have a number of methods associated with them automatically. These methods are generated by the data type classes.

For example, if we define a class Person with three properties:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
Property DOB As %Date;
}
```

The names of each generated method is the property name concatenated with the name of the method from the inherited class. For example, some of the methods associated with the DOB property are:

ObjectScript

```
Set x = person.DOBIsValid(person.DOB)
Write person.DOBLogicalToDisplay(person.DOB)
```

where **IsValid** is a method of the property class and **LogicalToDisplay** is a method of the %Date data type class.

4.7 Methods

There are two kinds of methods: instance methods and class methods (called static methods in other languages). In most cases, a method is a procedure.

4.7.1 Specifying Method Keywords

In a method definition, you can include optional compiler keywords that affect how the method behaves. The following list shows some of the most commonly seen method keywords:

ProcedureBlock

By default, the variables used in a method are private to that method, because by default all methods are [procedure blocks](#). To define a method as a non-procedure block, specify the ProcedureBlock keyword as 0. For example:

Class Member

```
Method MyMethod() [ ProcedureBlock = 0 ]
{
    //implementation details
}
```

In this case, the variables in this method would be public variables.

Language

You have the choice of implementation language when creating a server-side method in Caché. The options are `basic` (Caché Basic), `cache` (ObjectScript), `mvbasic` (MVBasic), and `tsql` (TSQL).

By default, a method uses the language specified by the [Language](#) keyword specified for the class. In most cases, that keyword is `cache` (ObjectScript).

Private

Specifies that the method is private. Subclasses inherit the value of the Private keyword and cannot override it.

By default, methods are public and can be accessed anywhere. You can mark a method as private (via the Private keyword). If you do:

- It can only be accessed by methods of the class to which it belongs.
- It does not appear in the InterSystems Class Reference, which is introduced later in “[InterSystems Class Reference](#).”

It is, however, inherited and available in subclasses of the class that defines the method.

Other languages often call such methods *protected methods*.

4.7.2 References to Other Class Members

Within a method, use the syntax shown here to refer to other class members:

- To refer to a parameter, use an expression like this:

```
..#PARAMETERNAME
```

In classes provided by InterSystems, all parameters are defined in all capitals, by convention, but your code is not required to do this.

- To refer to another method, use an expression like this:

```
..methodname(arguments)
```

Note that you cannot use this syntax within a class method to refer to an instance method.

- (Within an instance method only) To refer to a property of the instance, use an expression like this:

```
..PropertyName
```

Similarly, to refer to a property of an object-valued property, use an expression like this:

```
..PropertyNameA.PropertyNameB
```

This is known as *Caché dot syntax*.

Also, you can invoke an instance method of an object-valued property. For example:

```
do ..PropertyName.MyMethod( )
```

4.7.3 References to Methods of Other Classes

Within a method (or within a routine), use the syntax shown here to refer to a method in some other class:

- To invoke a class method and access its return value, use an expression like the following:

```
##class(Package.Class).MethodName(arguments)
```

For example:

ObjectScript

```
Set x=##class(Util.Utils).GetToday()
```

Or, if you are not interested in the return value, use **DO** as follows:

ObjectScript

```
Do ##class(Util.Utils).DumpValues()
```

Note: `##class` is not case-sensitive.

- To invoke an instance method, create an instance (as described in the next chapter) and then use an expression like the following to invoke the method and access its return value:

```
instance.MethodName(arguments)
```

For example:

ObjectScript

```
Set x=instance.GetName()
```

Or, if you are not interested in the return value, use **DO** as follows:

ObjectScript

```
Do instance.InsertItem("abc")
```

Not all methods have return values, so choose the syntax appropriate for your case.

4.7.4 References to Current Instance

Within an instance method, sometimes it is necessary to refer to the current instance itself, rather than to a property of method of the instance. For example, you might need to pass the current instance as an argument when invoking some other code. In such a case, use the special variable **\$THIS** to refer to the current instance.

For example:

ObjectScript

```
Set sc=header.ProcessService($this)
```

4.7.5 Method Arguments

A method can take positional arguments in a comma-separated list. For each argument, you can specify a type and the default value.

For instance, here is the partial definition of a method that takes three arguments:

Class Member

```
Method Calculate(count As %Integer, name, state As %String = "CA") as %Numeric
{
    // ...
}
```

Notice that two of the arguments have explicit types, and one has an default value. Generally it is a good idea to explicitly specify the type of each argument.

4.7.5.1 Skipping Arguments

In ObjectScript, when you invoke a method, you can skip arguments, if there are suitable defaults for them. For example, the following is valid:

ObjectScript

```
set myval=##class(mypackage.myclass).GetValue(,,,,,4)
```

4.7.5.2 Passing Variables by Value or by Reference

When you invoke a method, you can pass values of variables to that method either by value or by reference, in just the same way that you do with routines and subroutines; see “[Passing Variables by Value or by Reference](#),” earlier in this book:

The signature of a method usually indicates whether you are intended to pass arguments by reference. For example:

```
Method MyMethod(argument1, ByRef argument2, Output argument3)
```

The ByRef keyword indicates that you should pass this argument by reference. The Output keyword indicates that you should pass this argument by reference and that the method ignores any value that you initially give to this argument.

Similarly, when you define a method, you use the ByRef and Output keywords in the method signature to inform other users of the method how it is meant to be used.

Important: The ByRef and Output keywords provide information for the benefit of anyone using the InterSystems Class Reference, [introduced later](#). They do not affect the behavior of the code. It is the responsibility of the writer of the method to enforce any rules about how the method is to be invoked.

4.7.5.3 Variable Numbers of Arguments

You can define a method so that it accepts a variable number of arguments. For example:

Class Member

```

ClassMethod MultiArg(Arg1... As %List)
{
  Write "Invocation has ",
    $GET(Arg1, 0),
    " element",
    $SELECT(($GET(Arg1, 0)=1):"", 1:"s"),
    !
  For i = 1 : 1 : $GET(Arg1, 0)
  {
    Write:($DATA(Arg1(i))>0) "Argument[" , i , "]:",
      ?15, $GET(Arg1(i), "<NULL>"), !
  }
  Quit
}

```

Because methods are procedures, they support the `...` syntax to accept variable numbers of arguments. This syntax is described in the “[Variable Numbers of Arguments](#)” section of the “User-defined Code” chapter of *Using Caché ObjectScript*.

4.7.5.4 Specifying Default Values

To specify an argument’s default value, use syntax as shown in the following example:

Class Member

```

Method Test(flag As %Integer = 0)
{
  //method details
}

```

When a method is invoked, it uses its default values (if specified) for any missing arguments.

Another option is to use the `$GET` function. For example:

Class Member

```

Method Test(flag As %Integer)
{
  set flag=$GET(flag,0)
  //method details
}

```

This technique, however, does not affect the class signature.

4.8 Special Kinds of Methods

The `CodeMode` keyword enables you to define other, special kinds of methods:

4.8.1 Call Methods

A call method is a special mechanism to create method wrappers around existing Caché routines. The syntax for a call method is as follows:

Class Member

```

Method Call() [ CodeMode = call ]
{
  Label^Routine
}

```

where “Label^Routine” specifies a label within a routine.

4.8.2 Method Generators

A method generator is a program that is invoked by the class compiler during class compilation. Its output is the actual runtime implementation of the method. Method generators provide a means of inheriting methods that can produce high performance, specialized code that is customized to the needs of the inheriting class or property. Within the Caché library, method generators are used extensively by the data type and storage classes.

For details, refer to the “[Method Generators](#)” chapter of *Using Caché ObjectScript*.

4.9 Class Queries

A Caché class can contain class queries. A *class query* defines an SQL query that can be used by the class and specifies a class to use as a container for the query. The following shows an example:

Class Member

```
Query QueryName(Parameter As %String) As %SQLQuery
{
SELECT MyProperty, MyOtherProperty FROM MyClass
  WHERE (MyProperty = "Hello" AND MyOtherProperty = :Parameter)
  ORDER BY MyProperty
}
```

You define class queries to provide predefined lookups for use in your application. For example, you can look up instances by some property, such as by name, or provide a list of instances that meet a particular set of conditions, such as all the flights from Paris to Madrid. The example shown here uses a parameter, which is a common way to provide a flexible query. Note that you can define class queries within any class; there is no requirement to include class queries within persistent classes, which are introduced later in this book.

4.10 XData Blocks

Because XML is often a useful way to represent structured data, Caché classes include a mechanism that allow you to include well-formed XML documents, for any need you might have. To do this, you include an *XData block*, which is another kind of class member. The following shows an example:

The following shows an example:

```
XData Contents [XMLNamespace="http://www.intersystems.com/zen"]
{
  <page xmlns="http://www.intersystems.com/zen" title="HelpDesk">
    <html id="title">My Title</html>
    <hgroup>
      <pane paneName="menuPane"/>
      <spacer width="20"/>
      <vgroup width="100%" valign="top">
        <pane paneName="tablePane"/>
        <spacer height="20"/>
        <pane paneName="detailPane"/>
      </vgroup>
    </hgroup>
  </page>
}
```

Caché uses XData blocks for certain specific purposes, and these might give you ideas for your own applications:

- In Zen pages, you use XData blocks to describe the look and feel of the page.
- WS-Policy support for Caché web services and web clients. See [Creating Web Services and Web Clients in Caché](#). In this case, an XData block describes the security policy.
- In DeepSee, you use XData blocks to define cubes, subject areas, KPIs, and other elements.

Note that in all these cases, the XData block must be included within a class of a specific type.

4.11 Macros and Include Files in Class Definitions

In a Caché class definition, you can define macros in a method and use them in that method. More often, however, you define them in an [include file](#), which you can include at the start of any class definition. For example:

```
Include (%assert, %callout, %occInclude, %occSAX)

/// Implements an interface to the XSLT Parser. XML contained in a file or binary stream
/// may be transformed
Class %XML.XSLT.Transformer Extends %RegisteredObject ...
```

Then your methods in that class can refer to any macros defined in that include file, or in its included include files.

Macros are inherited. That is, a subclass has access to all the same macros as its superclasses.

4.12 Inheritance Rules in Caché

As with other class-based languages, you can combine multiple class definitions via inheritance. A Caché class definition can *extend* (or *inherit from*) multiple other classes. Those classes, in turn, can extend other classes.

The following subsections provide the basic rules for inheritance of classes in Caché.

4.12.1 Inheritance Order

Caché uses the following rules for inheritance order:

1. By default, if a class member of a given name is defined in multiple superclasses, the subclass takes the definition from the left-most class in the superclass list.
2. If the class definition contains `Inheritance = right`, then the subclass takes the definition from the right-most class in the superclass list.

For reasons of history, most Caché classes contain `Inheritance = right`.

4.12.2 Primary Superclass

Any class that extends other classes has a single *primary superclass*.

No matter which inheritance order a class uses, the primary superclass is the first one, reading left to right.

For any class-level compiler keywords, a given class uses the values specified in its primary superclass.

For a persistent class, the primary superclass is especially important; see “[Classes and Extents](#),” later in this book.

4.12.3 Most-Specific Type Class

Although an object can be an instance belonging to the extents of more than one class — such as that of various superclasses — it always has a *most-specific type class (MSTC)*. A class is the most specific type of an object when that object is an instance of that class, but is not an instance of any subclass of that class.

4.12.4 Overriding Methods

A class inherits methods (both class and instance methods) from its superclass or superclasses, which you can override. If you do so, you must ensure that the signature in your method definition matches the signature of the method you are overriding. This even includes that any argument of a subclass's method cannot have a data type specified if the matching argument of the superclass's method has no data type specified. The method in the subclass can, however, specify additional arguments that are not defined in the superclass.

Within a method in a subclass, you can refer to the method that it overrides in a superclass. To do so, use the `##super()` syntax. For example:

```
//overrides method inherited from a superclass
Method MyMethod()
{
    //execute MyMethod as implemented in the superclass
    do ##super()
    //do more things....
}
```

Note: `##super` is not case-sensitive.

4.13 For More Information

For more information on the topics covered in this chapter, see the following books:

- [Using Caché Objects](#) describes how to define classes and class members in Caché.
- [Caché Class Definition Reference](#) provides reference information for the compiler keywords that you use in class definitions.
- The InterSystems Class Reference, which is introduced later in “[InterSystems Class Reference](#),” provides details on all non-internal classes provided with Caché.

5

Caché Objects

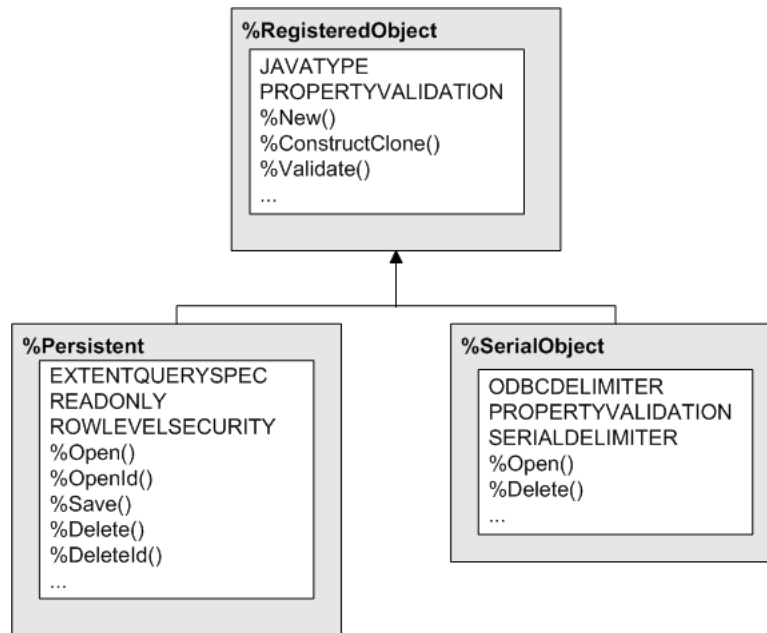
This chapter discusses objects in Caché. It discusses the following topics:

- [Introduction to Caché object classes](#)
- [Basic features of object classes](#)
- [OREFs](#)
- [Stream classes](#)
- [Collection classes](#)
- [Functions that are especially useful with classes](#)
- [Other sources of information on these topics](#)

5.1 Introduction to Caché Object Classes

Caché provides object technology by means of the following object classes: %Library.RegisteredObject, %Library.Persistent, and %Library.SerialObject.

The following figure shows the inheritance relationships among these classes, as well as some of their parameters and methods. The names of classes of the %Library package can be abbreviated, so that (for example) %Persistent is an abbreviation for %Library.Persistent. Here, the items in all capitals are parameters and the items that start with percent signs are methods.



In a typical class-based application, you define classes based on these classes (and on specialized system subclasses). All objects inherit directly or indirectly from one of these classes, and every object is one of the following types:

- A *registered object* is an instance of %RegisteredObject or a subclass. You can create these objects but you cannot save them. The other two classes inherit from %RegisteredObject and thus include all the parameters, methods, and so on of that class.
- A *persistent object* is an instance of %Persistent or a subclass. You can create, save, open, and delete these objects.
A persistent class is automatically projected to a table that you can access via Caché SQL. The [following chapter](#) discusses persistent classes in detail.
- A *serial object* is an instance of %SerialObject or a subclass. A serial class is meant for use as a property of another object. You can create and delete these objects, but you cannot save them or open them independently of the object that contains them.

When contained in persistent objects, these objects have an automatic projection to SQL.

Note: Via the classes %DynamicObject and %DynamicArray, Caché also provides the ability to work with objects and arrays that have no schema. These classes are not discussed in this book. For details, see [Using JSON in Caché](#).

5.2 Basic Features of Object Classes

With the object classes, you can perform the following tasks, among others:

- You can create an object (an *instance* of a class). To do so, you use the **%New()** method of that class, which it inherits from %RegisteredObject.

For example:

ObjectScript

```
set myobj=##class(Sample.Person).%New()
```

- You can use properties.

You can define properties in any class, but they are useful only in object classes, because only these classes enable you to create instances.

Any property contains a single literal value, an object (possibly a collection object), or a multidimensional array (rare). The following example shows the definition of an object-valued property:

Class Member

```
Property Home As Sample.Address;
```

`Sample.Address` is another class. The following shows one way to set the value of the `Home` property:

```
Set myaddress=##class(Sample.Address).%New()
Set myaddress.City="Louisville"
Set myaddress.Street="15 Winding Way"
Set myaddress.State="Georgia"

Set myperson=##class(Sample.Person).%New()
Set myperson.Home=myaddress
```

- You can invoke methods of an instance of the class, if the class or its superclasses define instance methods. For example:

Class Member

```
Method PrintPerson()
{
    Write !, "Name: ", ..Name
    Quit
}
```

If `myobj` is an instance of the class that defines this method, you can invoke this method as follows:

ObjectScript

```
Do myobj.PrintPerson()
```

- You can validate that the property values comply with the rules given in the property definitions.
 - All objects inherit the instance method **%NormalizeObject()**, which normalizes all the object's property values. Many data types allow different representations of the same value. Normalization converts a value to its canonical, or normalized, form. **%NormalizeObject()** returns true or false depending on the success of this operation.
 - All objects inherit the instance method **%ValidateObject()**, which returns true or false depending on whether the property values comply with the property definitions.
 - All persistent objects inherit the instance method **%Save()**. When you use the **%Save()** instance method, the system automatically calls **%ValidateObject()** first.

In contrast, when you work at the routine level and do not use classes, your code must include logic to check the type and other input requirements.

- You can define *callback methods* to add additional custom behavior when objects are created, modified, and so on.

For example, to create an instance of a class, you invoke the **%New()** method of that class. If that class defines the **%OnNew()** method (a *callback method*), then Caché automatically also calls that method. The following shows a simple example:

Class Member

```
Method %OnNew() As %Status
{
    Write "hi there"
    Quit $$$OK
}
```

In realistic scenarios, this callback might perform some required initialization. It could also perform logging by writing to a file or perhaps to a global.

5.3 OREFs

The **%New()** method of an object class creates an internal, in-memory structure to contain the object's data and returns an *OREF* (*object reference*) that points to that structure. An OREF is a special kind of value in ObjectScript. You should remember the following points:

- In the Terminal, when you display an OREF, you see a string that consists of a number, followed by an at sign, followed by the name of the class. For example:

```
SAMPLES>set myobj=##class(Sample.Person).%New()
SAMPLES>w myobj
3@Sample.Person
```

- Caché returns an error if you do not use an OREF where an OREF is expected:

```
SAMPLES>set myobj.Name="Fred Parker"
SET myobj.Name="Fred Parker"
^
<INVALID OREF>
```

Similarly:

```
SAMPLES>do myobj.PrintPerson()
DO myobj.PrintPerson()
^
<INVALID OREF>
```

It is helpful to be able to recognize this error. It means that the variable is not an OREF but should be.

- There is only one way to create an OREF: Use a method that returns an OREF. The methods that return OREFs are defined in the object classes or their subclasses.

The following does not create an OREF, but rather a string that *looks* like an OREF:

```
SAMPLES>set testthis="4@Sample.Person"
```

- You can determine programmatically whether a variable contains an OREF. The function **\$IsObject** returns 1 (true) if the variable contains an OREF; and it returns 0 (false) otherwise.

Note: For persistent classes, described in the [next chapter](#), methods such as **%OpenId()** also return OREFs.

5.4 Stream Interface Classes

As [noted earlier](#), Caché allocates a fixed amount of space to hold the results of string operations. If a string expression exceeds the amount of space allocated, a <MAXSTRING> error results. Unless [long strings](#) are enabled, this limit is 32 KB and no string property can be larger than about 32 KB. (No other property can exceed this limit, either, but other restrictions take effect before the long string limit applies.)

If you need to pass a long string value, or you need a property to contain a long string value, you use a stream. A *stream* is an object that can contain a single value whose size is larger than the string size limit. (Internally Caché creates and uses a temporary global to avoid the memory limitation.)

You can use stream fields with Caché SQL, with some restrictions. For details and a more complete introduction, see [Using Caché Objects](#); also see the InterSystems Class Reference for these classes.

5.4.1 Stream Classes

The main Caché stream classes use a common stream interface defined by the %Stream.Object class. You typically use streams as properties of other objects, and you save those objects. Stream data may be stored in either an external file or a Caché global, depending on the class you choose:

- The %Stream.FileCharacter and %Stream.FileBinary classes are used for streams written to external files.
(Binary streams contain the same kind of data as type %Binary, and can hold large binary objects such as pictures. Character streams contain the same kind of data as type %String, and are intended for storing large amounts of text.)
- The %Stream.GlobalCharacter and %Stream.GlobalBinary classes are used for streams stored in globals.

To work with a stream object, you use its methods. For example, you use the **Write()** method of these classes to add data to a stream, and you use **Read()** to read data from it. The stream interface includes other methods such as **Rewind()** and **MoveTo()**.

5.4.2 Example

For example, the following code creates a global character stream and writes some data into it:

ObjectScript

```
Set mystream=##class(%Stream.GlobalCharacter).%New()
Do mystream.Write("here is some text to store in the stream ")
Do mystream.Write("here is some more text")
Write "this stream has this many characters: ",mystream.Size,!
Write "this stream has the following contents: ",!
Write mystream.Read()
```

5.5 Collection Classes

When you need a container for sets of related values, you can use \$LIST format lists and multidimensional arrays, as described earlier in this book.

If you prefer to work with classes, Caché provides list classes and array classes; these are called *collections*.

5.5.1 List and Array Classes for Use As Standalone Objects

To create list objects, you can use the following classes:

- `%Library.ListOfDataTypes` — Defines a list of literal values.
- `%Library.ListOfObjects` — Defines a list of objects (persistent or serial).

To manipulate a list object, use its methods. For example:

ObjectScript

```
Set Colors=##class(%Library.ListOfDataTypes).%New()
Do Colors.Insert("Red")
Do Colors.Insert("Green")
Do Colors.Insert("Blue")
Write "Number of items in this list: ", Colors.Count()
Write !, "Second item in the list: ", Colors.GetAt(2)
```

Similarly, to create array objects, you can use the following classes:

- `%Library.ArrayOfDataTypes` — Defines an array of literal values. Each array item has a key and a value.
- `%Library.ArrayOfObjects` — Defines an array of objects (persistent or serial). Each array item has a key and an object value.

To manipulate an array object, use its methods. For example:

ObjectScript

```
Set ItemArray=##class(%Library.ArrayOfDataTypes).%New()
Do ItemArray.SetAt("example item","alpha")
Do ItemArray.SetAt("another item","beta")
Do ItemArray.SetAt("yet another item","gamma")
Do ItemArray.SetAt("still another item","omega")
Write "Number of items in this array: ", ItemArray.Count()
Write !, "Item that has the key gamma: ", ItemArray.GetAt("gamma")
```

5.5.2 List and Arrays as Properties

You can also define a property as a list or array.

To define a property as a list, use the following form:

Class Member

```
Property MyProperty As list of Classname;
```

If *Classname* is a data type class, then Caché uses the interface provided by `%Collection.ListOfDT`. If *Classname* is an object class, then it uses the interface provided by `%Collection.ListOfObj`.

To define a property as an array, use the following form:

Class Member

```
Property MyProperty As Array of Classname;
```

If *Classname* is a data type class, then Caché uses the interface provided by `%Collection.ArrayOfDT`. If *Classname* is an object class, then it uses the interface provided by `%Collection.ArrayOfObj`.

5.6 Useful ObjectScript Functions

ObjectScript provides the following functions for use with object classes:

- **\$CLASSMETHOD** enables you to run a class method, given as class name and method name. For example:

```
SAMPLES>set class="Sample.Person"
SAMPLES>set obj=$CLASSMETHOD(class,"%OpenId",1)
SAMPLES>w obj.Name
Van De Griek,Charlotte M.
```

This function is useful when you need to write generic code that executes a class method, but the class name (or even the method name) is not known in advance. For example:

ObjectScript

```
//read name of class from imported document
Set class=$list(headerElement,1)
// create header object
Set headerObj=$classmethod(class,"%New")
```

The other functions are useful in similar scenarios.

- **\$METHOD** enables you to run an instance method, given an instance and a method name. For example:

```
SAMPLES>set obj=##class(Sample.Person).%OpenId(1)
SAMPLES>do $METHOD(obj,"PrintPerson")
Name: Van De Griek,Charlotte M.
```

- **\$PROPERTY** gets or sets the value of the given property for the given instance. For example:

```
SAMPLES>set obj=##class(Sample.Person).%OpenId(2)
SAMPLES>write $property(obj,"Name")
Edison,Patrick J.
```

- **\$PARAMETER** gets the value of the given class parameter, given an instance. For example:

```
SAMPLES>set obj=##class(Sample.Person).%OpenId(2)
SAMPLES>write $parameter(obj,"EXTENTQUERYSPEC")
Name,SSN,Home.City,Home.State
```

- **\$CLASSNAME** returns the class name for a given instance. For example:

```
SAMPLES>set obj=##class(Sample.Person).%OpenId(1)
SAMPLES>write $CLASSNAME(obj)
Sample.Person
```

With no argument, this function returns the class name of the current context. This can be useful in instance methods.

5.7 For More Information

For more information on the topics covered in this chapter, see the following books:

- [*Using Caché Objects*](#) describes how to define classes and class members in Caché.

- [Caché Class Definition Reference](#) provides reference information for the compiler keywords that you use in class definitions.
- The InterSystems Class Reference, which is introduced later in “[InterSystems Class Reference](#),” provides details on all non-internal classes provided with Caché.

6

Persistent Objects and Caché SQL

A key feature in Caché is its combination of object technology and SQL. You can use the most convenient access mode for any given scenario. This chapter describes how Caché provides this feature and gives an overview of your options for working with stored data. It discusses the following topics:

- [Introduction](#)
- [Caché SQL](#)
- [Options for persistent classes](#)
- [SQL projection of persistent classes](#)
- [Object IDs](#)
- [Storage](#)
- [How to create classes and tables](#)
- [How to access stored data](#)
- [A look at stored data](#)
- [Other sources of information on these topics](#)

6.1 Introduction

Caché provides what is sometimes called an *object database*: a database combined with an object-oriented programming language. As a result, you can write flexible code that does all of the following:

- Perform a bulk insert of data via SQL.
- Open an object, modify it, and save it, thus changing the data in one or more tables without using SQL.
- Create and save new objects, adding rows to one or more tables without using SQL.
- Use SQL to retrieve values from a record that matches your given criteria, rather than iterating through a large set of objects.
- Delete an object, removing records from one or more tables without using SQL.

That is, you can choose the access mode that suits your needs at any given time.

Internally, all access is done via direct global access, and you can access your data that way as well when appropriate. (If you have a class definition, it is not recommended to use direct global access to make changes to the data.)

6.2 Caché SQL

Caché provides an implementation of SQL, known as Caché SQL.

Caché SQL supports the complete entry-level SQL-92 standard with a few exceptions and several special extensions. Caché SQL also supports indices, triggers, BLOBs, and stored procedures (these are typical RDBMS features but are not part of the SQL-92 standard). For a complete list, see [Using Caché SQL](#).

6.2.1 Where You Can Use Caché SQL

You can use Caché SQL within routines and within methods. To use SQL in these contexts, you can use either or both of the following tools:

- *Embedded SQL*, as in the following example:

ObjectScript

```
&sql(SELECT COUNT(*) INTO :myvar FROM Sample.Person)
Write myvar
```

You can use embedded SQL in ObjectScript routines and in methods written in ObjectScript.

- *Dynamic SQL* (the %SQL.Statement and %SQL.StatementResult classes), as in the following example:

```
SET myquery = "SELECT TOP 5 Name,DOB FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatus = tStatement.%Prepare(myquery)
SET rset = tStatement.%Execute()
//now use proprties of rset object
```

You can use dynamic SQL in any context.

Also, you can execute Caché SQL directly within the SQL Shell (in the Terminal) and in the Management Portal. Each of these includes an option to view the query plan, which can help you identify ways to make a query more efficient.

6.2.2 Object Extensions to SQL

To make it easier to use SQL within object applications, Caché includes a number of object extensions to SQL.

One of the most interesting of these extensions is ability to follow object references using the reference (“->”) operator. For example, suppose you have a Vendor class that refers to two other classes: Contact and Region. You can refer to properties of the related classes using the reference operator:

SQL

```
SELECT ID,Name,ContactInfo->Name
FROM Vendor
WHERE Vendor->Region->Name = 'Antarctica'
```

Of course, you can also express the same query using SQL JOIN syntax. The advantage of the reference operator syntax is that it is succinct and easy to understand at a glance.

6.3 Special Options for Persistent Classes

In Caché, all persistent classes extend `%Library.Persistent` (also referred to as `%Persistent`). This class provides much of the framework for the object-SQL correspondence in Caché. Within persistent classes, you have options like the following:

- Methods to open, save, and delete objects.

When you open a persistent object, you specify the degree of *concurrency locking*, because a persistent object could potentially be used by multiple users or multiple processes.

When you open an object instance and you refer to an object-valued property, the system automatically opens that object as well. This process is referred to as *swizzling* (also known as *lazy loading*). Then you can work with that object as well. For example:

ObjectScript

```
Set person=##class(Sample.Person).%OpenId(10)
Set person.Name="Andrew Park"
Set person.Address.City="Birmingham"
Do person.%Save()
```

Similarly, when you save an object, the system automatically saves all its object-valued properties as well; this is known as a *deep save*. There is an option to perform a *shallow save* instead.

- Default query (the Extent query) that is an SQL result set that contains the data for the objects of this class.

In this class (or in other classes), you can define additional queries; see “[Class Queries](#),” earlier in this book.

- Ability to define relationships between classes that are projected to SQL as foreign keys.

A relationship is a special type of object-valued property that defines how two or more object instances are associated with each other. Every relationship is two-sided: for every relationship definition, there is a corresponding inverse relationship that defines the other side. Caché automatically enforces referential integrity of the data, and any operation on one side is immediately visible on the other side. Relationships automatically manage their in-memory and on-disk behavior. They also provide superior scaling and concurrency over object collections (see “[Collection Classes](#)” in the previous chapter).

- Ability to define foreign keys. In practice, you add foreign keys to add referential integrity constraints to an existing application. For a new application, it is simpler to define relationships instead.
- Ability to define indices in these classes.

Indices provide a mechanism for optimizing searches across the instances of a persistent class; they define a specific sorted subset of commonly requested data associated with a class. They are very helpful in reducing overhead for performance-critical searches.

Indices can be sorted on one or more properties belonging to their class. This allows you a great deal of specific control of the order in which results are returned.

In addition, indices can store additional data that is frequently requested by queries based on the sorted properties. By including additional data as part of an index, you can greatly enhance the performance of the query that uses the index; when the query uses the index to generate its result set, it can do so without accessing the main data storage facility.

- Ability to define triggers in these classes to control what occurs when rows are inserted, modified, or deleted.
- Ability to project methods and class queries as SQL stored procedures.
- Ability to fine-tune the projection to SQL (for example, specifying the table and column names as seen in SQL queries).
- Ability to fine-tune the structure of the globals that store the data for the objects.

6.4 SQL Projection of Persistent Classes

For any persistent class, each instance of the class is available as a row in a table that you can query and manipulate via SQL. To demonstrate this, this section uses the Management Portal and the Terminal, which are introduced later in this book.

6.4.1 Demonstration of the Object-SQL Projection

Consider the `Sample.Person` class in `SAMPLES`. If we use the Management Portal to display the contents of the table that corresponds to this class, we see something like the following:

Refresh Close Window

Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-8538		Pueblo	AK	6977 First Street
2	2	17	05/30/1997	Green	Waterman,Danielle C.	944-39-5991		Oak Creek	ID	1648 Maple Street
3	3	86	04/01/1928		DeSantis,Christen N.	336-13-6311		Boston	AZ	8572 Maple Street
4	4	55	02/29/1960	Purple	Baker,Marvin Z.	198-22-7709		Queensbury	NV	1243 First Blvd
5	5	80	12/13/1934	Black	Diavolo,Ralph A.	586-13-9662		Hialeah	NY	3880 Maple Plaza
6	6	38	10/13/1976		Russell,Paul S.	572-40-8824		Denver	CA	7269 Main Plaza
7	7	33	11/26/1981	Purple Purple	Pascal,John X.	468-82-7179		Zanesville	AR	872 Elm Street

(This is not the same data that you see, because this sample is repopulated at each release.) Note the following points:

- The values shown here are the display values, not the logical values as stored on disk.
- The first column (**#**) is the row number in this displayed page.
- The second column (**ID**) is the unique identifier for a row in this table; this is the identifier to use when opening objects of this class. (In this class, these identifiers are integers, but that is not always true.)

These numbers happen to be the same in this case because this table is freshly populated each time the `SAMPLES` database is built. In a real application, it is possible that some records have been deleted, so that there are gaps in the **ID** values and these values do not match the row numbers.

In the Terminal, we can use a series of commands to look at the first person:

```
SAMPLES>set person=##class(Sample.Person).%OpenId(1)
SAMPLES>write person.Name
Van De Griek,Charlotte M.
SAMPLES>write person.FavoriteColors.Count()
1
SAMPLES>write person.FavoriteColors.GetAt(1)
Red
SAMPLES>write person.SSN
571-15-2479
```

These are the same values that we see via SQL.

6.4.2 Basics of the Object-SQL Projection

Because inheritance is not part of the relational model, the class compiler projects a “flattened” representation of a persistent class as a relational table. The following table lists how some of the various object elements are projected to SQL:

Object Concept	SQL Concept
Package	Schema
Class	Table
Data type property	Field
Embedded object	Set of fields
List property	List field
Array property	Child table
Stream property	BLOB
Index	Index
Class method marked as stored procedure	Stored procedure

The projected table contains all the appropriate fields for the class, including those that are inherited.

6.4.3 Classes and Extents

Caché uses an unconventional and powerful interpretation of the object-table mapping.

All the stored instances of a persistent class compose what is known as the *extent* of the class, and an instance belongs to the extent of *each* class of which it is an instance. Therefore:

- If the persistent class `Person` has the subclass `Student`, the `Person` extent includes all instances of `Person` and all instances of `Student`.
- For any given instance of class `Student`, that instance is included in the `Person` extent and in the `Student` extent.

Indices automatically span the entire extent of the class in which they are defined. The indices defined in `Person` contain both `Person` instances and `Student` instances. Indices defined in the `Student` extent contain only `Student` instances.

The subclass can define additional properties not defined in its superclass. These are available in the extent of the subclass, but not in the extent of the superclass. For example, the `Student` extent might include the `FacultyAdvisor` field, which is not included in the `Person` extent.

The preceding points mean that it is comparatively easy in Caché to write a query that retrieves all records of the same type. For example, if you want to count people of all types, you can run a query against the `Person` table. If you want to count only students, run the same query against the `Student` table. In contrast, with other object databases, to count people of all types, it would be necessary to write a more complex query that combined the tables, and it would be necessary to update this query whenever another subclass was added.

6.5 Object IDs

Each object has a unique ID within each extent to which it belongs. In most cases, you use this ID to work with the object. This ID is the argument to the following commonly used methods of the `%Persistent` class:

- `%DeleteId()`
- `%ExistsId()`
- `%OpenId()`

The class has other methods that use the ID, as well.

6.5.1 How an ID Is Determined

Caché assigns the ID value when you first save an object. The assignment is permanent; you cannot change the ID for an object. Objects are not assigned new IDs when other objects are deleted or changed.

Any ID is unique within its extent.

The ID for an object is determined as follows:

- For most classes, by default, IDs are integers that are assigned sequentially as objects of that class are saved.
- For a class that is used as the child in a parent-child relationship, the ID is formed as follows:

```
parentID | childID
```

Where *parentID* is the ID of the parent object and *childID* is the ID that the child object would receive if it were not being used in a parent-child relationship. Example:

```
104 | 3
```

This ID is the third child that has been saved, and its parent has the ID 104 in its own extent.

- If the class has an index of type `IdKey` and the index is on a specific property, then that property value is used as the ID.

```
SKU-447
```

Also, the property value cannot be changed.

- If the class has an index of type `IdKey` and that index is on multiple properties, then those property values are concatenated to form the ID. For example:

```
CATEGORY12 | SUBCATEGORYA
```

Also, these property values cannot be changed.

6.5.2 Accessing an ID

To access the ID value of an object, you use the `%Id()` instance method that the object inherits from `%Persistent`.

In SQL, the ID value of an object is available as a pseudo-field called `%id`. Note that when you browse tables in the Management Portal, the `%id` pseudo-field is displayed with the caption ID:

Refresh

Close Window

Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First Street
2	2	17	05/30/1997	Green	Waterman,Danielle C.	944-39-5991		Oak Creek	ID	1648 Maple Street
3	3	86	04/01/1928		DeSantis,Christen N.	336-13-6311		Boston	AZ	8572 Maple Street
4	4	55	02/29/1960	Purple	Baker,Marvin Z.	198-22-7709		Queensbury	NV	1243 First Blvd
5	5	80	12/13/1934	Black	Diavolo,Ralph A.	586-13-9662		Hialeah	NY	3880 Maple Place
6	6	38	10/13/1976		Russell,Paul S.	572-40-8824		Denver	CA	7269 Main Plaza
7	7	33	11/26/1981	Purple Purple	Pascal,John X.	468-82-7179		Zanesville	AR	872 Elm Street

Despite this caption, the name of the pseudo-field is `%Id`.

6.6 Storage

Each persistent class definition includes information that describes how the class properties are to be mapped to the globals in which they are actually stored. The class compiler generates this information for the class and updates it as you modify and recompile.

6.6.1 A Look at a Storage Definition

It can be useful to look at this information, and on rare occasions you might want to change some of the details (very carefully). For a persistent class, Studio displays something like the following as part of your class definition:

```
<Storage name="Default">
<Data name="PersonDefaultData"><Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Name</Value>
</Value>
<Value name="3">
<Value>SSN</Value>
</Value>
<Value name="4">
<Value>DOB</Value>
</Value>
...
</Storage>
```

6.6.2 Globals Used by a Persistent Class

The storage definition includes several elements that specify the globals in which the data is stored:

```
<DataLocation>^Sample.PersonD</DataLocation>
<IdLocation>^Sample.PersonD</IdLocation>
<IndexLocation>^Sample.PersonI</IndexLocation>
...
<StreamLocation>^Sample.PersonS</StreamLocation>
```

By default, with default storage:

- The class data is stored in the *data global* for the class. Its name starts with the complete class name (including package name). A “D” is appended to the name. For example: `Sample.PersonD`
- The index data is stored in the *index global* for the class. Its name starts with the class name and ends with an “I”. For example: `Sample.PersonI`
- Any saved stream properties are stored in the *stream global* for the class. Its name starts with the class name and ends with an “S”. For example: `Sample.PersonS`

Important: If the complete class name is long, the system automatically uses a hashed form of the class name instead. So when you view a storage definition, you might sometimes see global names like `^package1.pC347.VeryLongCla4F4AD`. If you plan to work directly with the data global for a class for any reason, make sure to examine the storage definition so that you know the actual name of the global.

Informally, these globals are sometimes called the *class globals*, but this can be a misleading phrase. The class definitions are not stored in these globals.

For more information on how global names are determined, see “[Globals](#)” in *Using Caché Objects*.

6.6.3 Default Structure for a Stored Object

For a typical class, most data is contained in the data global, which includes nodes as follows:

Node	Node Contents
<code>^full_class_name("id")</code> Where <i>full_class_name</i> is the complete class name including package, hashed if necessary to keep the length to 31 characters. Also, <i>id</i> is the object ID as described in “ Object IDs .”	List of the format returned by \$ListBuild . In this list, stored properties are listed in the order given by the <code>name</code> attribute of the <code><Value></code> element in the storage definition. By definition, transient properties are not stored. Stream properties are stored in the stream global for the class.

For an example, see “[A Look at Stored Data](#),” later in this chapter.

6.6.4 Notes

Note the following points:

- Never redefine or delete storage for a class that has stored data. If you do so, you will have to recreate the storage manually, because the new default storage created when you next compile the class might not match the required storage for the class.
- During development, you may want to reset the storage definition for a class. You can do this if you also delete the data and later reload or regenerate it.

For details, see “[Useful Skills to Learn](#),” later in this book.

- By default, as you add and remove properties during development, the system automatically updates the storage definition, via a process known as *schema evolution*.

The exception is if you use a non-default storage class for the `<Type>` element. The default is `%Library.CacheStorage`; if you do not use this storage class, Caché does not update the storage definition. The other common option is `%Library.CacheSQLStorage`, which is used primarily to support applications written before Caché provided classes.

6.7 Options for Creating Persistent Classes and Tables

To create a persistent class and its corresponding SQL table, you can do any of the following:

- Use Studio to define a class based on `%Persistent`. When you compile the class, the system creates the table.
- In the Management Portal, you can use the Data Migration Wizard, which reads an external table, prompts you for some details, generates a class based on `%Persistent`, and then loads records into the corresponding SQL table.
You can run the wizard again later to load more records, without redefining the class.
- In the Management Portal, you can use the Link Table Wizard, which reads an external table, prompts you for some details, and generates a class that is linked to the external table. The class retrieves data at runtime from the external table.

This is a special case and is not discussed further in this book.

- In the Management Portal, you can use the FileMan Wizard, which reads FileMan files and creates classes.

- In Caché SQL, use CREATE TABLE or other DDL statements. This also creates a class.
- In the Terminal (or in code), use the **CSVTOCLASS()** method of %SQL.Util.Procedures. For details, see the Class Reference for %SQL.Util.Procedures.

6.8 Accessing Data

To access, modify, and delete data associated with a persistent class, your code can do any or all of the following:

- Open instances of persistent classes, modify them, and save them.
- Delete instances of persistent classes.
- Use embedded SQL.
- Use dynamic SQL (the SQL statement and result set interfaces).
- Use low-level commands and functions for direct global access. Note that this technique is not recommended *except* for retrieving stored values, because it bypasses the logic defined by the object and SQL interfaces.

Caché SQL is suitable in situations like the following:

- You do not initially know the IDs of the instances to open but will instead select an instance or instances based on input criteria.
- You want to perform a bulk load or make bulk changes.
- You want to view data but not open object instances.
(Note, however, that when you use object access, you can control the degree of concurrency locking. If you know that you do not intend to change the data, you can use minimal concurrency locking.)
- You are fluent in SQL.

Object access is suitable in situations like the following:

- You are creating a new object.
- You know the ID of the instance to open.
- You find it more intuitive to set values of properties than to use SQL.

6.9 A Look at Stored Data

This section demonstrates that for any persistent object, the same values are visible via object access, SQL access, and direct global access.

In Studio, if we view the `Sample.Person` class, we see the following property definitions:

```
/// Person's name.
Property Name As %String(POPSPEC = "Name()") [ Required ];

...

/// Person's age.<br>
/// This is a calculated field whose value is derived from <property>DOB</property>.
Property Age As %Integer [ details removed for this example ];

/// Person's Date of Birth.
Property DOB As %Date(POPSPEC = "Date()");
```

In the Terminal, we can open a stored object and write its property values:

```
SAMPLES>set person=##class(Sample.Person).%OpenId(1)

SAMPLES>w person.Name
Newton,Dave R.
SAMPLES>w person.Age
14
SAMPLES>w person.DOB
58153
```

Note that here we see the literal, stored value of the DOB property. We could instead call a method to return the display value of this property:

```
SAMPLES>write person.DOBLogicalToDisplay(person.DOB)
03/20/2000
```

In the Management Portal, we can browse the stored data for this class, which looks as follows:

Refresh Close Window

Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First Street
2	2	17	05/30/1997	Green	Waterman,Danielle C.	944-39-5991		Oak Creek	ID	1648 Maple Street
3	3	86	04/01/1928		DeSantis,Christen N.	336-13-6311		Boston	AZ	8572 Maple Street
4	4	55	02/29/1980	Purple	Baker,Marvin Z.	198-22-7709		Queensbury	NV	1243 First Blvd
5	5	80	12/13/1934	Black	Diavolo,Ralph A.	586-13-9662		Hialeah	NY	3880 Maple Pl
6	6	38	10/13/1976		Russell,Paul S.	572-40-8824		Denver	CA	7269 Main Pl
7	7	33	11/26/1981	Purple Purple	Pascal,John X.	468-82-7179		Zanesville	AR	872 Elm Street

Notice that in this case, we see the display value for the DOB property. (In the Portal, there is another option to execute queries, and with that option you can control whether to use logical or display mode for the results.)

In the Portal, we can also browse the global that contains the data for this class:

Global Search Mask: Display Cancel

Search History: Maximum Rows: 100 ☐ Allow Edit

1:	^Sample.PersonD	= 200
2:	^Sample.PersonD(1)	= \$lb("", "Newton,Dave R.", "384-10-6538", 58153, \$lb("6977 First Street", "Pueblo", "AK"
3:	^Sample.PersonD(2)	= \$lb("", "Waterman,Danielle C.", "944-39-5991", 57128, \$lb("1648 Maple Street", "Oak C
4:	^Sample.PersonD(3)	= \$lb("", "DeSantis,Christen N.", "336-13-6311", 31867, \$lb("8572 Maple Street", "Bostc
5:	^Sample.PersonD(4)	= \$lb("", "Baker,Marvin Z.", "198-22-7709", 43523, \$lb("1243 First Blvd", "Queensbury",

Or, in the Terminal, we can write the value of the global node that contains this instance:

```
zw ^Sample.PersonD("1")
^Sample.PersonD(1)=$lb( " ", "Newton,Dave R. ", "384-10-6538", 58153, $lb( "6977 First
Street", "Pueblo", "AK", 63163),
$lb( "9984 Second Blvd", "Washington", "MN", 42829), " ", $lb( "Red" ) )
```

For reasons of space, the last example contains an added line break.

6.10 Storage of Generated Code for Caché SQL

For Caché SQL (except when used as embedded SQL), the system generates reusable code to access the data.

When you first execute an SQL statement, Caché optimizes the query and generates and stores code that retrieves the data. It stores the code in the *query cache*, along with the optimized query text. Note that this cache is a cache of code, not of data.

Later when you execute an SQL statement, Caché optimizes it and then compares the text of that query to the items in the query cache. If Caché finds a stored query that matches the given one (apart from minor differences such as whitespace), it uses the code stored for that query.

You can view the query cache and delete any items in it.

6.11 For More Information

For more information on the topics covered in this chapter, see the following:

- “[Useful Skills to Learn](#),” later in this book, has information on viewing data and the query cache.
- [Using Caché Objects](#) describes how to define classes and class members in Caché.
- [Caché Class Definition Reference](#) provides reference information for the compiler keywords that you use in class definitions.
- [Using Caché SQL](#) describes how to use Caché SQL and where you can use it.
- [Caché SQL Reference](#) provides reference information on Caché SQL.
- [Using Caché Globals](#) provides details on how Caché stores persistent objects in globals.
- The InterSystems Class Reference, which is introduced later in “[InterSystems Class Reference](#),” has information on all non-internal classes provided by Caché.

7

Namespaces and Databases

This chapter describes how Caché organizes data and code. It discusses the following:

- [Introduction](#)
- [Database basics](#)
- [System-supplied databases](#)
- [System-supplied namespaces](#)
- [Custom items in CACHESYS](#)
- [What is accessible in your namespaces](#)
- [Stream directory for a namespace](#)
- [Other sources of information on these topics](#)

The chapter “[Useful Skills to Learn](#)” includes information on configuring namespaces, defining mappings, moving code and data, and so on.

7.1 Introduction to Namespaces and Databases

In Caché, any code runs in a *namespace*, which is a logical entity. A namespace provides access to data and to code, which is stored (typically) in multiple databases. A *database* is a file — a CACHE.DAT file. Caché provides a set of namespaces and databases for your use, and you can define additional ones.

In a namespace, the following options are available:

- A namespace has a default database in which it stores code; this is the *routine database* for this namespace.
When you write code in a namespace, the code is stored in its routine database unless other considerations apply. Similarly, when you invoke code, Caché looks for it in this database unless other considerations apply.
- A namespace also has a default database to contain data for persistent classes and any globals you create; this is the *global database* for this namespace.
So, for example, when you access data (in any manner), Caché retrieves it from this database unless other considerations apply.
The global database can be the same as the routine database, but it is often desirable to separate them for maintainability.
- A namespace has a default database for temporary storage.

- A namespace can include *mappings* that provide access to additional data and code that is stored in other databases. Specifically, you can define mappings that refer to routines, class packages, entire globals, and specific global nodes in non-default databases. (These kinds of mappings are called, respectively, *routine mappings*, *package mappings*, *global mappings*, and *subscript-level mappings*.)

When you provide access to a database via a mapping, you provide access to only a part of that database. The namespace cannot access the non-mapped parts of that database, not even in a read-only manner.

Also, it is important to understand that when you define a mapping, that affects only the configuration of the namespace. It does not change the current location of any code or data. Thus when you define a mapping, it is also necessary to move the code or data (if any exists) from its current location to the one expected by the namespace.

Defining mappings is a database administration task and requires no change to class/table definitions or application logic.

The chapter “[Useful Skills to Learn](#)” provides further details.

- Any namespace you create has access to most of the Caché code library. This code is available because Caché automatically establishes specific mappings for any namespace you create.

This book has mentioned some of these classes; to find tools for a particular purpose, see the *InterSystems Programming Tools Index*.

Mappings provide a convenient and powerful way to share data and code. Any given database can be used by multiple namespaces. For example, there are several system databases that all customer namespaces can access, as discussed later in this chapter.

You can change the configuration of a namespace after defining it, and Caché provides tools for moving code and data from one database to another. Thus you can reorganize your code and data during development, if you discover the need to do so. This makes it possible to reconfigure Caché applications (such as for scaling) with little effort.

7.1.1 Locks, Globals, and Namespaces

Because a global can be accessed from multiple namespaces, Caché provides automatic cross-namespace support for its locking mechanism. A lock on a given global applies automatically to *all* namespaces that use the database that stores the global.

For an introduction to locking, see “[Locking and Concurrency Control](#),” earlier in this book.

7.2 Database Basics

A Caché *database* is a CACHE.DAT file. You create a database via the Management Portal. Or if you have an existing Caché database, you can configure Caché to become aware of it.

7.2.1 Database Configuration

For any database, Caché requires the following configuration details:

- Logical name for the database.
- Directory in which the CACHE.DAT file resides. When you create a database in the Management Portal, you are prompted to choose or create a subdirectory within the *system manager’s directory* (*cache-install/Mgr*), but you can store the database file in any convenient directory.

Tip: It is convenient to use the same string for the logical name and for the directory that contains the CACHE.DAT file. The system-provided Caché databases follow this convention.

Additional options include the following:

- Default directory to use for file streams used by this database.

This is important because your users will need write access to this directory; if not, your code will not be able to create file streams.

- Collation of new globals.
- Initial size and other physical characteristics.
- Option to enable or disable *journaling*. Journaling tracks changes made to a Caché database, for up-to-the-minute recovery after a crash or restoring your data during system recovery.

In most cases, it is best to enable journaling. However, you might want to disable journaling for designated temporary work spaces; for example, the CACHETEMP database is not journaled.

- Option to mount this database for read-only use.

If a user tries to set a global in a read-only database, Caché returns a <PROTECT> error.

In most cases, you can create, delete, or modify database attributes while the system is running.

7.2.2 Database Features

With each database, Caché provides physical integrity guarantees for both the actual data and the metadata that organizes it. This integrity is guaranteed even if an error occurs during writes to the database.

The databases are automatically extended as needed, without manual intervention. If you expect a particular database to grow and you can determine how large it will become, you can “pre-expand” it by setting its initial size to be near the expected eventual size. If you do so, the performance is better.

Caché provides a number of strategies that allow high availability and recoverability. These include:

- Journaling — Introduced earlier.
- Mirroring — Provides rapid, reliable, robust, automatic failover between two Caché systems, making mirroring the ideal automatic failover high-availability solution for the enterprise.
- Shadowing — Using this facility, one Caché database server “shadows” another by reading the primary server’s journal and applying database changes to its own copy of the database. Note that this is not true replication, but allows for emergency failover where a small amount of latency is permissible.
- Clustering — There is full support of clustering on operating systems that provide it.

Caché has a technology for distributing data and application logic and processing among multiple systems. It is called the Enterprise Cache Protocol (ECP). On a multiserver system, a network of Caché database servers can be configured as a common resource, sharing data storage and application processing, with the data distributed seamlessly among them. This provides increased scalability as well as automatic failover and recovery.

7.2.3 Database Portability

Caché databases are portable across platforms and across versions, with the following caveats:

- On different platforms, any file is either *big-endian* (that is, most-significant byte first) or *little-endian* (least-significant byte first).

Caché provides a utility to convert the byte order of a Caché database; it is called *cvendian*. This is useful when moving a database among platforms of the two types. For details, see the section “[Using cvendian to Convert Between Big-endian and Little-endian Systems](#)” in *Caché Specialized System Tools and Utilities*.

- If you use a Unicode version of Caché to create a database, you might lose data if you try to use that database with an 8-bit Caché installation, because that installation cannot retrieve 16-bit character data.

You can use an 8-bit database with a Unicode installation, however.

- If you use an 8-bit version of Caché, your data is not portable to an 8-bit locale that uses a different character set.
- If a database contains code from an earlier release of Caché, you can use that code with a later version of Caché.

In some cases, it is necessary to make changes. InterSystems strives to assure that applications written for a version of Caché will run without change on subsequent versions, but there are exceptions.

- If a database contains code or data that was created when long strings were enabled, then you might not be able to use that database with a Caché system in which long strings are not enabled. See the chapter “[Server Configuration Options](#),” later in this book.

7.3 System-Supplied Databases

Caché provides the following databases:

- **CACHELIB** — This is a read-only database that includes the object, data type, stream, and collection classes and many other class definitions. It also includes the system include files, generated INT code (for most classes), and generated OBJ code.
- **CACHESYS** (the *system manager’s database*) — This database includes utilities and data related to system management. It is intended to contain specific custom code and data of yours and to preserve that code and data upon upgrades.

This database contains or can contain:

- Users, roles, and other security elements (both predefined items and ones that you add).
For reasons of security, the Management Portal handles this data differently than other data; for example, you cannot display a table of users and their passwords.
- Data for use by the NLS (National Language Support) classes: number formats, the sort order of characters, and other such details. You can load additional data.
- Your own code and data. To ensure that these items are preserved upon upgrades, use the naming conventions in “[Custom Items in CACHESYS](#),” later in this chapter.

CAUTION: InterSystems does not support moving, replacing, or deleting this database.

The directory that contains this database is the *system manager’s directory*. The console log (*cconsole.log*) is written to this directory, as are other log files.

- **CACHE** — Contains items such as cached SQL queries and CSP session information.

Note: No customer application should directly interact with the CACHE database.

- **CACHEAUDIT** — When you enable event logging, Caché writes the audit data to this database.
- **CACHETEMP** — Caché uses this for temporary storage, and you can use it as well for the same purpose. This database is reinitialized each time the system restarts; see also the [MaxCacheTempSizeAtStart](#) CPF keyword. Note that this database is not journaled.

- **DOCBOOK** — This database contains the documentation that the DocBook application serves up to viewers.
- **SAMPLES** — This database contains code samples.
- **USER** — This database is empty and is reserved for your use. Unlike the other databases, this database is entirely untouched by the installer, after the initial installation.

For additional detail on CACHESYS and SAMPLES, see the chapter “[Assets and Resources](#)” in the *Caché Security Administration Guide*.

7.4 System-Supplied Namespaces

Caché provides the following namespaces for your direct use:

- **%SYS** — This namespace provides access to code that should not be available in all namespaces — code that manipulates security elements, the server configuration, and so on.

For this namespace, the default routine database and default global database is CACHESYS. If you follow certain naming conventions, you can create your own code and globals in this namespace and store it in that database. See the [next section](#).

- **DOCBOOK** — The DocBook application uses this namespace.
- **SAMPLES** — This namespace provides access to the SAMPLES database.
- **USER** — This namespace provides access to the USER database.

Caché uses other namespaces for its own purposes; see “[Configuring Namespaces](#)” in the *Caché System Administration Guide*.

7.5 Custom Items in CACHESYS

You can create items in the CACHESYS database. When you install a Caché upgrade, this database is upgraded. During this upgrade, some items are deleted unless they follow the naming conventions for custom items.

To add code or data to this database so that your items are not overwritten, do one of the following:

- Go to the %SYS namespace and create the item. For this namespace, the default routine database and default global database are both CACHESYS. Use the following naming conventions to prevent your items from being affected by the upgrade installation:
 - **Classes:** start the package with Z or z
 - **Routines:** start the name with Z, z, %Z, or %z
 - **Globals:** start the name with ^Z, ^z, ^%Z, or ^%z
- In *any* namespace, create items with the following names:
 - **Routines:** start the name with %Z or %z
 - **Globals:** start the name with ^%Z or ^%z

Note: InterSystems reserves the use of some %Z* routine names, including: %ZEN*, %ZHSLIB*.

Because of the standard mappings in any namespace, these items are written to CACHESYS.

MAC code and include files are not affected by upgrade.

7.6 What Is Accessible in Your Namespaces

When you create a namespace, the system automatically defines mappings for that namespace. As a result, in that namespace, you can use the following items (provided you are logged in as a user with suitable permissions for these items):

- Any class whose package name starts with a percent sign (%). This includes most, but not all, classes provided by Caché.
- All the code stored in the routine database for this namespace.
- All the data stored in the global database for this namespace.
- Any routine whose name starts with a percent sign.
- Any include file whose name starts with a percent sign.
- Any global whose name starts with a caret and a percent sign (^%). These globals are generally referred to as *percent globals*. Note that via global mappings or subscript level mappings, it is possible to change where percent globals are stored, but that has no effect on their visibility. Percent globals are always visible in all namespaces.
- Your own globals with names that start **^CacheTempUser** — for example, `^CacheTempUser.MyApp`. If you create such globals, these globals are written to the CACHETEMP database.
- Any additional code or data that is made available via mappings defined in this namespace.

Via extended global references, your code can access globals that are defined in other namespaces. For information, see “[Global Structure](#)” in *Using Caché Globals*.

The Caché security model controls which data and which code any user can access; see the chapter “[Caché Security](#)” for an introduction.

7.6.1 System Globals in Your Namespaces

Your namespaces contain additional system globals, which fall into two rough categories:

- System globals that are in all namespaces. These include the globals in which Caché stores your routines, class definitions, include files, INT code, and OBJ code.
- System globals that are created when you use specific Caché features. For example, if you use DeepSee in a namespace, the system creates a set of globals for DeepSee to use.

In most cases, you should not manually write to or delete any of these globals. See “[Global Naming Conventions](#),” in *Using Caché Globals*.

7.7 Stream Directory

In any given namespace, when you create a file stream, Caché writes a file to a default directory and then later deletes it.

This is important because your users will need write access to this directory; if not, your code will not be able to create file streams.

The default directory is the stream subdirectory of the global database for this namespace.

7.8 For More Information

For more information on the topics covered in this chapter, see the following:

- “[Useful Skills to Learn](#),” later in this book, includes information on defining namespaces and databases.
- *[Using Caché Globals](#)* has information on working with globals, including using extended references.
- “[Using cvendian to Convert Between Big-endian and Little-endian Systems](#)” in *[Caché Specialized System Tools and Utilities](#)* has information on cvendian.
- *[Caché High Availability Guide](#)* has information on high availability and recoverability.
- *[Caché Distributed Data Management Guide](#)* has information on Enterprise Cache Protocol (ECP).

8

Caché Security

This chapter provides an overview of Caché security, with emphasis on the topics most relevant to programmers who write or maintain Caché applications. It discusses the following topics:

- [Introduction to Caché security](#)
- [Caché applications](#)
- [Authorization in Caché](#)

Security is discussed in detail in the *Caché Security Administration Guide*.

8.1 Introduction

This section provides an introduction to security within Caché and for communications between Caché and external systems.

8.1.1 Security Elements Within Caché

Caché security provides a simple, unified security architecture that is based on the following elements:

- **Authentication.** Authentication is how you prove to Caché that you are who you say you are. Without trustworthy authentication, authorization mechanisms are moot — one user can impersonate another and then take advantage of the fraudulently obtained privileges.

The authentication mechanisms available depend on how you are accessing Caché. Caché has a number of available authentication mechanisms. Some require programming effort.

- **Authorization.** Once a user is authenticated, the next security-related question to answer is what that person is allowed to use, view, or alter. This determination and control of access is known as *authorization*.

As a programmer, you are responsible for including the appropriate security checks within your code to make sure that a given user has permission to perform a given task. The authorization model is discussed in more detail later in this chapter.

- **Auditing.** Auditing provides a verifiable and trustworthy trail of actions related to the system, including actions of the authentication and authorization systems. This information provides the basis for reconstructing the sequence of events after any security-related incident. Knowledge of the fact that the system is audited can serve as a deterrent for attackers (because they know they will reveal information about themselves during their attack).

Caché provides a set of events that can be audited, and you can add others. As a programmer, you are responsible for include the audit logging in your code for your custom events.

- *Database encryption.* Caché database encryption protects data at rest — it secures information stored on disk — by preventing unauthorized users from viewing this information. Caché implements encryption using the AES (Advanced Encryption Standard) algorithm. Encryption and decryption occur when Caché writes to or reads from disk. In Caché, encryption and decryption have been optimized, and their effects are both deterministic and small for any Caché platform; in fact, there is no added time at all for writing to an encrypted database.

The task of database encryption does not generally require you to write code.

8.1.2 Secure Communications to and From Caché

When communicating between Caché and external systems, you can use the following additional Caché tools:

- *SSL/TLS configurations.* Caché supports the ability to store a SSL/TLS configuration and specify an associated name. When you need an SSL/TLS connection (for HTTP communications, for example), you programmatically provide the applicable configuration name, and Caché automatically handles the SSL/TLS connection.
- *X.509 certificate storage.* Caché supports the ability to load an X.509 certificate and private key and specify an associated configuration name. When you need an X.509 certificate (to digitally sign a SOAP message, for example), you programmatically provide the applicable configuration name, and Caché automatically extracts and uses the certificate information.

You can optionally enter the password for the associated private key file, or you can specify this at runtime.

- *Access to a certificate authority (CA).* If you place a CA certificate of the appropriate format in the prescribed location, Caché uses it to validate digital signatures and so on.

Caché uses the CA certificate automatically; no programming effort is required.

8.2 Caché Applications

Almost all users interact with Caché using *applications*. For example, the Management Portal itself is a set of applications. Each application has its own security. There are three kinds of applications in Caché:

- *Web applications* — these are applications built with the InterSystems tools for front-end development: CSP and Zen. To create these, you create the web pages and you use the Management Portal to register the pages (starting with a particular root path) formally as an application.
- *Privileged routine applications* — these are routines in any supported server-side language. In addition to writing the routine, you use the Management Portal to register it formally as an application.
- *Client applications* — these are applications that use [Caché Direct](#) to connect to Caché. (For information on Caché Direct, see [Using Caché Direct](#).) In addition to creating the program, you use the Management Portal to register it formally as an application.

You can define, modify, and applications within the Management Portal (provided that you are logged in as a user with sufficient privileges). When you deploy your applications, however, you are more likely to define applications programmatically as part of installation; Caché provides ways to do so.

8.3 Caché Authorization Model

As a programmer, you are responsible for including the appropriate security checks within your code to make sure that a given user has permission to perform a given task. Therefore, it is necessary to become familiar with the Caché authorization model, which uses role-based access. Briefly, the terms are as follows:

- *Assets*. Assets are the items being protected. Assets vary widely in nature. The following items are all assets:
 - Each Caché database
 - The ability to connect to Caché using SQL
 - The ability to perform backups
 - Each DeepSee KPI class
 - Each application defined in Caché
- *Resources*. A resource is a Caché security element that you can associate with one or more assets.
 For some assets, the association between an asset and a resource is a configuration option. When you create a database, you specify the associated resource. Similarly, when you create a Caché application, you specify the associated resource.
 For other assets, the association is hardcoded. For a DeepSee KPI class, you specify the associated resource as a parameter of that class.
 For assets and resources that you define, you are free to make the association in either manner — either by hardcoding it or by defining a suitable configuration system.
- *Roles*. A role is a Caché security element that specifies a name and an associated set of privileges (possibly quite large). A *privilege* is a *permission* of a specific type (Read, Write, or Use) on a specific resource. For example, the following are privileges:
 - Permission to read a database
 - Permission to write to a table
 - Permission to use an application
- *Username*s. A username (or a *user*, for short) is a Caché security element with which a user logs on to Caché. Each user *belongs to* (or *is a member of*) one or more roles.

Another important concept is *role escalation*. Sometimes it is necessary to temporarily add one or more new roles to a user (programmatically) so that the user can perform a normally disallowed task within a specific context. This is known as *role escalation*. After the user exits that context, you would remove the temporary roles; this is *role de-escalation*.

You define, modify, and delete resources, roles, and users within the Management Portal (provided that you are logged in as a user with sufficient privileges). When you deploy your applications, however, you are more likely to define resources, roles, and starter usernames programmatically, as part of installation; Caché provides ways to do so.

9

Localization Support

This chapter provides an overview of Caché support for localization. It discusses the following topics:

- [Introduction to localization in Caché](#)
- [Caché locales](#)
- [Default I/O translation tables](#)
- [Files and character encoding](#)
- [How to manually translate characters](#)

9.1 Introduction

Caché supports localization so that you can develop applications for multiple countries or multiple areas without needing to re-engineer the application. The Caché localization model works as follows:

- Caché provides a set of predefined locales. A Caché *locale* is a set of metadata that specify the user language, currency symbols, formats, and other conventions for a specific country or geographic region. The next section of this chapter provides more details.

The locale specifies the character encoding to use when writing to the Caché database. It also includes information necessary to handle character conversions to and from other character encodings.

- When you install a Caché server, the installer sets the default locale for that server.

This cannot be changed after installation, but you can specify that a process uses a non-default locale, if wanted.

- The Management Portal displays strings in the local language as specified by the browser settings, for a fixed set of languages.
- You can provide localized strings for your own applications as well. See the article [String Localization and Message Dictionaries](#). This mechanism is available for REST services, CSP pages, Zen pages, and DeepSee elements.

9.2 Caché Locales and National Language Support

A Caché *locale* is a set of metadata that defines storage and display conventions that apply to a specific country or geographic region. The locale definition includes the following:

- Number formats
- Date and time formats
- Currency symbols
- The sort order of words
- The default character set (the character encoding of this locale), as defined by a standard (ISO, Unicode, or other).

Note that Caché uses the phrases *character set* and *character encoding* as though they are synonymous, which is not strictly true in all cases.

- A set of *translation tables* (also called *I/O tables*) that convert characters to and from other supported character sets.
- The “translation table” for a given character set (for example, CP1250) is actually a pair of tables. One table specifies how to convert from the default character set to the foreign character set, and other specifies how to convert in the other direction. In Caché, the convention is to refer to this pair of tables as a single unit.

Caché uses the phrase *National Language Support* (NLS) to refer collectively to the locale definitions and to the tools that you use to view and extend them.

The Management Portal provides a page where you can see the default locale, view the details of any installed locale, and work with locales. The following shows an example:

Locale properties of enuw (English, United States, Unicode):
Your current locale is: enuw (English, United States, Unicode)
(enuw is a system locale. Edit is not allowed.)

Basic Properties

Name	Value
Country	United States (US)
Language	English (en-US)
Character set	Unicode
Currency	\$

You can also use this page to see the names of the available translation tables. These names are specific to Caché. (In some cases, discussed later in this chapter, it is necessary to know the names of these tables.)

For information on accessing and using this Management Portal page, see “[Using the NLS Pages of the Management Portal](#)” in the *Caché System Administration Guide*.

Caché also provides a set of classes (in the %SYS.NLS and Config.NLS packages). See “[System Classes for National Language Support](#)” in the chapter “Customizing the Caché System” in *Caché Specialized System Tools and Utilities*.

9.3 Default I/O Tables

External to the definition of any locale, a given Caché instance is configured to use specific translation tables, by default, for input/output activity. Specifically, it specifies the default translation tables to use in the following scenarios:

- When communicating with a Caché process

- When communicating with a Terminal
- When communicating with another terminal
- When reading from and writing to files
- When reading from and writing to tapes
- When reading from and writing to TCP/IP devices
- When reading from and writing to strings sent to the operating system as parameters (such as file names and paths)
- When reading from and writing to printers

For example, when Caché needs to call an operating system function that receives a string as a parameter (such as a file name or path), it first passes the string through an NLS translation appropriately called `syscall`. The result of this translation is sent to the operating system.

To see the current defaults, use `%SYS.NLS.Table`; see the class reference for detail.

9.4 Files and Character Encoding

Whenever you read to or write from an entity external to the database, there is a possibility that the entity is using a different character set than Caché. The most common scenario is working with files.

At the lowest level, you use the [Open](#) command to open a file or other device. This command can accept a parameter that specifies the translation table to use when translating characters to or from that device. For details, see the [Caché I/O Device Guide](#). Then Caché uses that table to translate characters as needed.

Similarly, when you use the object-based file APIs, you specify the `TranslateTable` property of the file.

(Note that the Ensemble adapter classes instead provide properties to specify the foreign character set — to be used as the expected character encoding system of input data and the desired character encoding of output data. In this case, you specify a standard character set name, choosing from the set supported by InterSystems.)

9.5 Manually Translating Characters

Caché provides the [\\$ZCONVERT](#) function, which you can use to manually translate characters to or from another character set. For example:

ObjectScript

```
IF $SYSTEM.Version.IsUnicode() {
    SET greek=$CHAR(945,946,947,913,914,915)
    WRITE $ZCONVERT(greek,"W")
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```


10

Development Tools

This chapter introduces the tools you commonly use as a developer. It discusses the following:

- [Management Portal](#)
- [Studio](#)
- [Terminal](#)
- [System qualifiers and flags](#)
- [InterSystems classes, routines, bindings, and other elements](#)
- [InterSystems Class Reference](#)
- [Tools for debugging](#)
- [Other sources of information on these topics](#)

10.1 Management Portal

The Management Portal is a browser-based tool that you can use to manage data and to configure Caché. System administrators use it later for other tasks.

10.1.1 Accessing the Portal

You can start the Management Portal in the following ways:

- From your web browser, go to the following URL:

`http://localhost:57772/csp/sys/UtilHome.csp`

Where *localhost* is the IP address of your system and 57772 is the port number of the web server installed by Caché.

- On Microsoft Windows platforms, select the InterSystems Launcher and then select **Management Portal**.

Important: *Microsoft Windows 2003 Users Trusted Site Security Setting* — The first time you visit the Management Portal, you may receive a warning about the web site being blocked. When prompted, add the site to the Trusted sites zone. InterSystems also recommends you allow session cookies for portal procedures to function properly.

Enter a Caché username and password if prompted.

The chapter “[Useful Skills to Learn](#)” describes many tasks you perform within this tool.

10.2 Studio

Studio is a tool that you use to build and debug code in all languages that Caché supports. It runs on Windows-based operating systems. It can connect to any Caché server (compatible with the current version of Studio) regardless of the platform and operating system of the server.

10.2.1 Starting Studio

To start Studio, select the InterSystems Launcher and then select **Studio**. Enter a Caché username and password if prompted.

Or use other options on the InterSystems Launcher to access remote systems. For details and additional options, see [Using the Studio](#).

You can open multiple Studios at the same time, which is useful if you are working in multiple namespaces.

10.2.2 Orientation

In Studio, you work within a single namespace at one time. To choose a namespace, select **File > Change Namespace** or press **F4**.

Studio displays all code to which this namespace has access; see “[What Is Accessible in Your Namespace](#),” earlier in this book. You cannot edit code that is stored in CACHELIB (which is read-only). Some classes provided by InterSystems are shipped in “deployed” mode, and you cannot see their internal details.

10.2.3 Documents (“Files”)

Studio manages units of code that it refers to as *files*; more accurately, they are simply *documents*, most of which are stored in the Caché databases used by the namespace that you are currently viewing. The **Workspace** window displays names of documents and groups them as follows:

- **Classes** folder — Lists class definitions, whose names have the form *package.classname.cls*
- **Routines** folder — Lists the following items:
 - ObjectScript routines, whose names have the form *name.mac*
 - Caché Basic routines, whose names have the form *name.bas*
 - Caché MVBasic routines, whose names have the form *name.MVB*
 - Intermediate code, whose names have the form *name.int* and *name.mvi* extensions, corresponding to INT code and MVI code respectively
 - Include files, whose names have the form *name.inc*
- **CSP Files** folder — Lists CSP (Caché Server Page) files, which have the form *csp.pagename.csp*.

Unlike the other documents, these actually reside outside the database, because the web server requires them to be there. When you open one of these in Studio, Studio retrieves the file from disk and displays it. When you save changes, Studio updates the file on disk.

Studio provides syntax coloring and checking for each of these kinds of documents.

If you have multiple Studios at the same time, you can drag a document from one Studio to another. This action imports the document into the other Studio (which is typically using a different namespace).

10.2.4 Integration with Source Control

All routines and class definitions are stored in a Caché database, which is different from most development toolkits. Studio provides a set of source control hooks which InterSystems developers use and which you can also use. See [Using Studio](#).

When you have these configured for your source control system, you can work with other developers like this:

- Each document is mapped to a file actually on your disk. The file contains an XML representation of the routine or class definition.
- When you start to make an edit to a routine or a class, Studio prompts you to check out that file from your source control system, and then communicates with the source control system as needed.
- When you save a change, Studio also exports the routine or class definition to the appropriate file.
- Studio provides a menu option to enable you to check the file in. This option sends a command to the source control system, which checks the file in and sends your changes to the source control server.
- Studio typically provides other menu options to interact with the source control system.

You might not need to use these source control hooks immediately, but you should be aware of them.

10.2.5 Other Environmental Options

It is useful to check the options used in Studio to see if they meet your current needs. Select **Tools > Options**. Some of the options include:

- Font typeface and size (in **Environment > Font**)
- Default display of storage (in **Environment > Class**)
- Display of generated INT code in the **Workspace** window, on the **Namespace** tab (in **Environment > Advanced**)
- Syntax checking options (in **Editor > Syntax Check and Assist**)
- Control of indentation (in **Editor > Indentation**)
- Compiler flags (in **Compiler > Flags & Optimization**)

Tip: **Keep Generated Source Code** controls whether the compiler keeps the INT code rather than discarding it after generating the OBJ code.

Also see “[System Qualifiers and Flags](#),” later in this chapter.

10.3 Terminal

The Terminal is a command-line interface for entering Caché commands and displaying current values. It is useful during learning, development, and debugging.

10.3.1 Starting the Terminal

To start the Terminal and use the local database, select the InterSystems Launcher and then select **Terminal**.

10.3.2 Orientation

In the Terminal, you work within a single namespace at one time. The prompt displayed in this window indicates the namespace in which you are currently working. For example:

```
USER>
```

To change to a new namespace, use the **ZNSPACE** command (which has a short form of **ZN**):

```
USER>ZN "SAMPLES"  
SAMPLES>
```

(Note that the **ZNSPACE** command is intended for use in the Terminal. When you change namespace within your code, InterSystems recommends that you use **NEW \$NAMESPACE** and **SET \$NAMESPACE** as described in the [\\$NAMESPACE](#) reference page.)

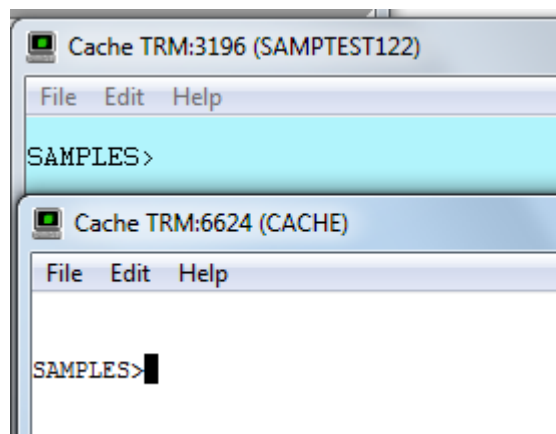
10.3.3 Environmental Options

You may find it useful to set up the Terminal for yourself as follows:

- Specify the namespace that it use every time you log in. To do so, specify the **Startup Namespace** option for the user that you used when you logged in. See the chapter “[Users](#)” in the *Caché Security Administration Guide*.
- Enable Windows-style edit accelerators. Select **Edit > User Settings** and select the **Windows edit accelerators** option. Select **Save** to save your selection for later sessions.
- Increase the window size. Select **Edit > Window Size** and specify the number of columns, rows, and scrollback lines. Select **Save** to save your selection for later sessions.
- Change the font size and other font options. Select **Edit > Font** and make changes. Select **OK** to save your selection for later sessions.
- Change the background and foreground colors. Select **Edit > Colors** and make changes. Select **Save** to save your selection for later sessions.

Then use **Edit > Erase** to refresh the Terminal window.

Tip: If you have multiple instances of Caché running on your machine, it is useful to customize the colors and fonts for the Terminal so that you can easily distinguish different Terminal windows. For example:



10.3.4 Using the Terminal

In the Terminal, you can enter most Caché commands (and you do not need to include a space at the start of a line). For example:

```
d ^myroutine

set dirname = "c:\test"

set obj=##class(Test.MyClass).%New()
write obj.Prop1
```

You cannot use labels, multiline constructs, or macros.

When you exit the Terminal, the system closes any open files and stops any foreground execution.

10.3.5 Common Keyboard Accelerators

It is useful to remember the basic keyboard accelerators:

- To interrupt the Terminal and stop any foreground execution, use one of the following key combinations:
 - **Ctrl+c** — Use this if the **Windows edit accelerators** option is not enabled.
 - **Ctrl+Shift+c** — Use this if the **Windows edit accelerators** option is enabled.

To find this option, select **Edit > User Settings**.

- To pause the Terminal scrolling, press **Ctrl+s**.

While the scrolling is paused, the Terminal accepts commands and processes them, but it does not write the commands or any output to the screen.

Tip: If the Terminal sometimes appears to be unresponsive, you may have accidentally pressed **Ctrl+s**.

- To resume, press **Ctrl+q**.
- To repeat a previous command, press the up arrow key repeatedly until the desired command is displayed. To enter the command, press **Enter**.

10.4 System Qualifiers and Flags

There are system qualifiers and flags that affect the import of external sources into Caché, the compilation process, and the export of code to external destinations. These settings are saved in a system global for each namespace, but you can provide overrides as follows:

- In Studio, you can specify settings that take precedence over what is stored in the system for this namespace. See “[Other Environmental Options](#),” earlier in this chapter.
- When you invoke a method to compile, export, and so on, you can specify the *qspec* argument, which is a concatenation of the flags and qualifiers you want to use. In this case, you are using methods in the class %SYSTEM.OBJ. See the [InterSystems Class Reference](#) for details.

10.4.1 Viewing the Current Default Qualifiers and Flags

To see the current flags and qualifiers used in a namespace, use the following commands:

ObjectScript

```
DO $system.OBJ.ShowFlags()
```

and

ObjectScript

```
DO $system.OBJ.ShowQualifiers()
```

These methods also display help information for the flags and qualifiers.

10.4.2 Changing the Defaults

You can change the defaults for the namespace in which you are working. To do so, use `$system.OBJ.SetFlags()` and `$system.OBJ.SetQualifiers()`. For example:

ObjectScript

```
DO $system.OBJ.SetFlags("ck")
DO $SYSTEM.OBJ.SetQualifiers("/skipstorage")
```

10.4.3 Key Flags

The following table lists a few key flags, to highlight them and to give you a sense of what the flags do:

Flag	Meaning	Default
b	Include subclasses and classes that reference the current class in SQL usage.	
c	Compile the class definitions after loading.	
e	Delete extent.	
k	Keep source. When this flag is set, source code of generated routines will be kept.	
l	Lock classes while compiling.	X
r	Recursive. Compile all the classes that are dependency predecessors.	
u	Update only. Skip compilation of classes that are already up-to-date.	

10.4.4 Historical Note

Flags have been available for longer. They were modeled on UNIX® command-line parameters and thus are one- or two-character sequences. The qualifiers were added later, as a richer, more extensible set of controls. To preserve backward compatibility, the flag mechanism remains fully supported. In addition, there is an existing qualifier to match each existing flag, and you can use the two in the same specifier.

10.5 InterSystems Classes and Routines

In addition to the basic classes introduced earlier (such as %RegisteredObject and %Persistent), Caché provides a wide set of useful classes that can reduce your development time if you choose to use them. For example, there are classes you can use for tasks such as these:

- Executing Caché SQL statements and working with result sets
- Working with files
- Creating and sending HTTP requests
- Working with XML files
- Programmatically compiling your code
- Managing system processes

Caché also provides routines for specific purposes (typically debugging or management), as well as language bindings and other tools. To find tools for a particular purpose, see the *InterSystems Programming Tools Index*.

10.6 InterSystems Class Reference (Documatic)

Caché provides a set of web pages called the *InterSystems Class Reference* or (informally) *Documatic*. These pages provide information on all classes provided by InterSystems, as well as all classes you create. The reference information shows the definitions of all class members, apart from their actual implementations. That is, you can see method signatures but not their internal definitions. It includes links between elements so that you can rapidly follow the logic of the code; in some cases, this is quicker than using Studio. There is also a search option.

Tip: To locate classes quickly, see the *InterSystems Programming Tools Index*. This book is organized into high-level topics of interest to programmers and provides an extensive set of links to specific classes in the *InterSystems Class Reference*.

10.6.1 Accessing the InterSystems Class Reference

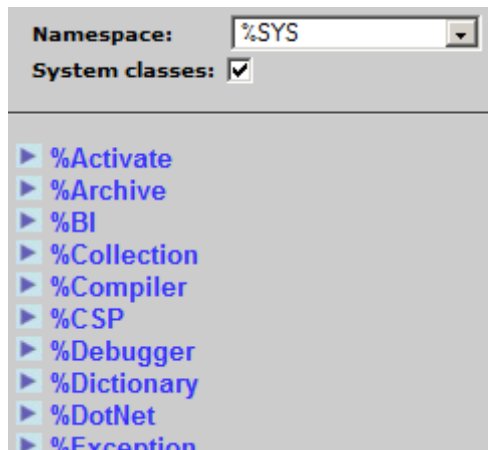
To view the InterSystems Class Reference, do any of the following:

- Start the online documentation system. On the main page, select **InterSystems Class Reference**.
- Enter the following URL into a browser page on the local InterSystems server, where 57772 is the web server port number configured for the server:
<http://localhost:57772/csp/documatic/%25CSP.Documatic.cls>
- In Studio, select **View > Show Class Documentation**.
- When viewing code in Studio, select the name of a class and press **F1**. The InterSystems Class Reference opens at the description of the class.

10.6.2 A Quick Look at the InterSystems Class Reference

Note: In these examples, to prevent confusion when you are reading this book online, the InterSystems Class Reference is shown with a gray background instead of the default color, which is white.

The following shows the left area, which you use for overall navigation:



Here you can do the following:

- Choose the namespace of interest.
- Hide or display the system classes.
- Expand packages, which displays their classes.
- Select a package or class to see details for it. The system then displays a reference page on the right.

The next picture shows the reference page for %Library.String:

Class Reference
%Library.String

DocBook | [Search](#)

[%SYS] > [%Library] > [String]

Server: **lhayden64**

Instance: **CACHE**

User: **UnknownUs**

☐ Private ☐ Storage

datatype class **%Library.String** extends [%DataType](#)

ODBC Type: **VARCHAR**

The **%String** data type class represents a string.

The logical value of the **%String** data type is a string.

▼ [Inventory](#)

Parameters	Properties	Methods	Queries	Indices	ForeignKeys	Triggers
10		5				

▼ [Summary](#)

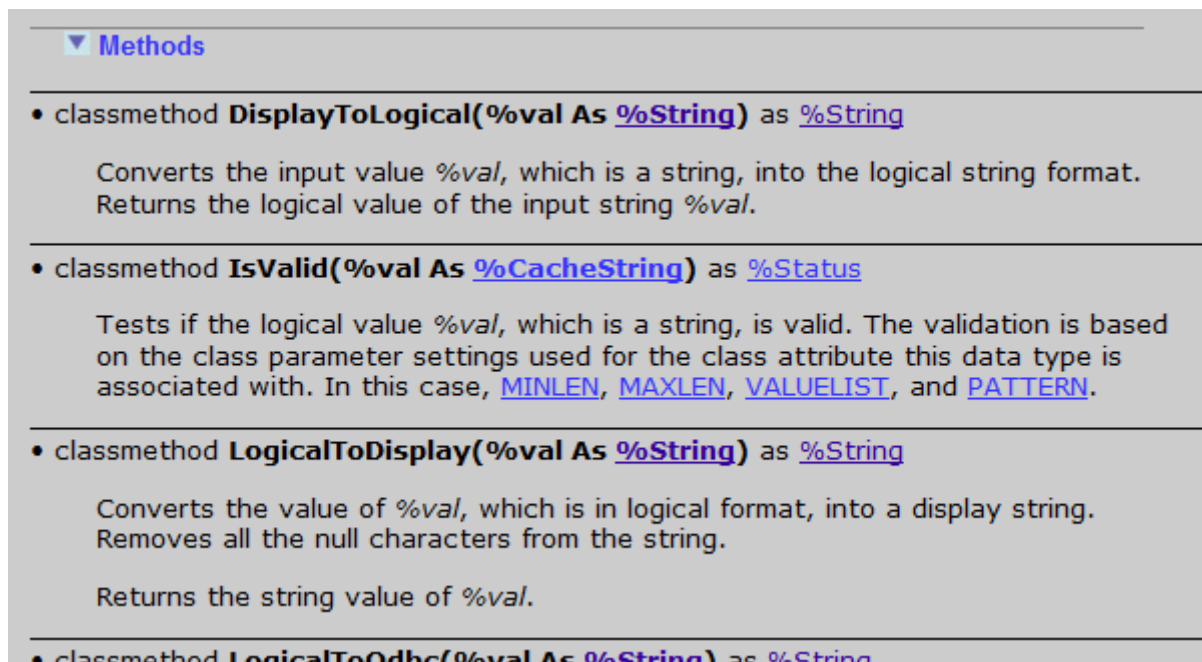
Methods			
DisplayToLogical	IsValid	LogicalToDisplay	LogicalToOdbc
Normalize			

Subclasses			
%Activate.SafeArray	%Activate.UserDefined	%BI.ConditionName	%CSP.Util.Choice
%CSP.Util.Passwd	%Identity.String	%Library.Char	%Library.ExactString
%Library.NetworkAddress	%Library.SysPath	%Library.Text	%Library.VarString

Notice the following:

- **DocBook** is a link to the DocBook application.
- **Search** displays a search page.
- The first line shows the class from which %Library.String inherits. This class name is a link to the reference page for that classes.
- The area below that shows the comments contained in the class definition, for the class itself. The system finds any comments that start with three slashes (///) and treats them as the description of the item that follows them.
- The **Summary** area shown here lists parameters, properties, and methods in this class. If you select any of these items, the page scrolls down to that element.
- If you select the **Private** check box, the page also displays any class members that are marked private. The Private keyword marks a class member as private, so that it can be used only by other members of the same class.
- If you select the **Storage** check box, the page also displays any storage information.

The lower area of the reference page shows details for the class members as in the following example:



If you select the links in the method signature or in the comments, the system displays the referenced class.

10.7 Tools for Debugging

This section summarizes the main tools for debugging in Caché.

- [Studio Debugger](#)
- [ZBREAK](#)
- [^%STACK](#)
- [Trace statements](#)
- [Audit log](#)

- [System logs](#)

Also see the chapter “[Command-line Routine Debugging](#)” in Using Caché ObjectScript, which lists some additional tools.

10.7.1 Studio Debugger

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

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

- Define a debug target by using **Debug > Debug Target**. Then select **Debug > Go** to start the target program and connect to its server process.
- Select **Debug > Attach** and choose a running process on a Caché server.

For details, see “[Using the Studio Debugger](#)” in Using Studio.

10.7.2 ZBREAK

Sometimes you have better control if you perform command-line debugging with the Caché Debugger and the **ZBREAK** command.

The Caché Debugger enables you to run debugging commands as if they were directly contained in your routine code. When you run the code, you can issue commands to test the conditions and the flow of processing within your application. Its major capabilities are:

- Set breakpoints with the **ZBREAK** command at code locations and take specified actions when those points are reached.
- Set watchpoints on local variables and take specified actions when the values of those variables change.
- Interact with Caché during a breakpoint/watchpoint in a separate window.
- Trace execution and output a trace record (to a terminal or other device) whenever the path of execution changes.
- Display the execution stack.
- Run an application on one device while debugging I/O goes to a second device. This enables full screen Caché applications to be debugged without disturbing the application’s terminal I/O.

For details, see “[Debugging with the Caché Debugger](#)” in Using Caché ObjectScript.

10.7.3 ^%STACK

The **^%STACK** routine lets you examine the process execution stack. You can use it to:

- Display the values of local variables, including values that have been “hidden” with the **NEW** command or through parameter passing.
- Display the values of process state variables, such as **\$IO** and **\$JOB**.

For details, see “[Using %STACK to Display the Stack](#)” in Using Caché ObjectScript.

10.7.4 Trace Statements

As with any programming language, you can edit the code to include statements that generate output about the current state of any variables of interest. In other languages, you might write output to the current device or print to a file, and you can do these things in Caché if desired.

Caché provides an alternative: set nodes of your own global. Use the **\$INCREMENT** function to create a set of nodes that have integer keys. For example:

ObjectScript

```
if (myval=mytestval) {
    //do stuff
    set ^CacheTempUserMyTrace($increment(^CacheTempUserMyTrace))="main branch"
} else {
    //do other stuff
    set ^CacheTempUserMyTrace($increment(^CacheTempUserMyTrace))="else branch"
}
```

Another possibility is as follows:

ObjectScript

```
set ^CacheTempUserMyTrace($increment(^CacheTempUserMyTrace))=$LISTBUILD("message",data)
```

For your global, use a name of the form `^CacheTempUser*`, which is written to the CACHETEMP database. Then if there is a transaction that gets rolled back, your global is not rolled back and thus contains an accurate record of the sequence of events.

10.7.5 Audit Log

When debugging code, sometimes it is useful to enable auditing and to view the *audit log*, if you have access to this administrative tool.

Caché allows you to monitor events and add entries to the audit database when those events occur. Auditable events fall into two categories:

- System events — Events that monitor actions within Caché, such as startup, shutdown, logins, and so on; system events also monitor security-related events, such as changes to security or audit settings. These events also include <PROTECT> errors and abnormal terminations.
- User events — Custom events that you define.

For information on enabling auditing and viewing the audit log, see the chapter “[Auditing](#)” in the *Caché Security Administration Guide*.

10.7.6 System Logs

The system manager’s directory, typically `install\dir\mgr`, contains several log files that may be helpful in debugging, including the console log (`cconsole.log`), System Monitor log (`SystemMonitor.log`), and initialization log (`cboot.log`). For information about these log files, see “[Monitoring Log Files](#)” in the *Caché Monitoring Guide*.

10.8 For More Information

For more information on the topics covered in this chapter, see the following:

- [*Caché System Administration Guide*](#) describes how to use most of the Management Portal.
- [*Using Studio*](#) describes how to use Studio and includes information on using its source control hooks.
- [*Using the Terminal*](#) describes how to use the Terminal, including how to launch different shells, such as the MV shell and the TSQL shell.
- [*Caché ObjectScript Reference*](#) provides reference information for the operators, commands, functions, special variables, and other parts of ObjectScript.
- [*InterSystems Programming Tools Index*](#) is a directory of the InterSystems classes, routines, bindings, and other programming tools, grouped by topic.

11

Server Configuration Options

There are a few configuration options for the server that can affect how you write your code. This chapter discusses the following topics:

- [Support for long string operations](#)
- [Settings for Caché SQL](#)
- [Use of IPv6 Internet addresses](#)
- [How to set these options programmatically](#)
- [Other sources of information on these topics](#)

Most of the configuration details are saved in a file called `cache.cpf` (the *CPF file*).

11.1 Support for Long String Operations

As noted earlier, Caché allocates a fixed amount of memory to hold the results of string operations. If a string expression exceeds the amount of memory allocated, a `<MAXSTRING>` error results.

- If long strings are not enabled, this limit is 32,767 characters.

For class definitions, this limit imposes a size limit on string properties. As noted earlier, when you need to work with strings that exceed this limit, you can use streams; see “[Stream Interface Classes](#).”

- If you enable long strings, then the limit is 3,641,144 characters.

For applications that use Caché MVBasic, you must enable long strings.

If long string operations are enabled:

- There is no intrinsic difference in disk space used apart from the size of the data itself.
- More memory is allocated. Whether the memory is actually used depends on two factors:
 - Whether the process actually performs an operation involving long strings.
 - How the operating system uses memory. Some operating systems use a lazy algorithm for memory allocation and will actually use virtual memory only when needed.

11.1.1 Enabling Long String Operations

To enable long strings:

1. Access the Management Portal.
2. Select **System Administration > Configuration > System Configuration > Memory and Startup**.
3. Check the value of the **Enable Long Strings** field. If this field is not checked, select it.
4. If you make any change, select **Save**.

11.2 Settings for Caché SQL

To view and modify the settings that affect the behavior of Caché SQL:

1. Access the Management Portal.
2. Select **System Administration > Configuration > SQL and Object Settings > General SQL Settings**.

This page lists many settings. The most important ones are typically these:

- **Cached Query - Save Source** — This specifies whether to save the routine and INT code that Caché generates when you execute any Caché SQL except for embedded SQL. (In all cases, the generated OBJ is kept. By default, the routine and INT code is not kept.)

The query results are not stored in the cache.

- **Default SQL Schema Name** — This specifies the default schema name to use when creating tables from external data. This schema name is used for any tables that do not have a specified schema.
- **Support Delimited Identifiers** — This controls how Caché SQL treats characters contained within a pair of double quotes.

If you enable support for delimited identifiers, you can use double quotes around the names of fields, which enables you to refer to fields whose names are not regular identifiers. Such fields might, for example, use SQL reserved words as names.

If you disable support for delimited identifiers, characters within double quotes are treated as string literals, and it is not possible to refer to fields whose names are not regular identifiers.

3. If you make any change, select **Save**.

11.3 Use of IPv6 Addressing

Caché always accepts IPv4 addresses and DNS forms of addressing (host names, with or without domain qualifiers). You can configure Caché to also accept IPv6 addresses; see “[IPv6 Support](#)” in the chapter “Configuring Caché” in the *Caché System Administration Guide*.

11.4 Configuring a Server Programmatically

You can programmatically change some of the operational parameters of Caché by invoking specific utilities; this is how you would likely change the configuration for your customers. For example:

- `Config.Miscellaneous` includes methods to set system-wide default and settings. One method, for example, enables long string operations.
- `%SYSTEM.Process` includes methods to set environment values for the life of the current process.
- `%SYSTEM.SQL` includes methods for changing SQL settings.

For details, see the InterSystems Class Reference for these classes.

11.5 For More Information

For more information on the topics covered in this chapter, see the following:

- [Caché System Administration Guide](#) describes how to use most of the Management Portal.
- The InterSystems Class Reference, which is introduced in “[InterSystems Class Reference](#),” provides details on all non-internal classes provided with Caché.
- [Caché Parameter File Reference](#) contains reference information on the CPF file.

12

Useful Skills to Learn

The purpose of this chapter is to briefly describe the Caché-specific tasks with which you should familiarize yourself. If you are familiar with how, when, and why to perform the tasks described in this chapter, you will be able to save yourself some time and effort. It discusses the following topics:

- [How to find the definition of a code element](#)
- [How to define databases](#)
- [How to define namespaces](#)
- [How to define mappings for globals](#)
- [How to define mappings for routines](#)
- [How to define mappings for packages](#)
- [How to generate test data](#)
- [How to remove stored data](#)
- [How to reset storage](#)
- [How to browse a table](#)
- [How to run an SQL query](#)
- [How to see object properties](#)
- [How to see the contents of a global](#)
- [How to view INT code](#)
- [How to view a query plan](#), which helps you determine how to speed up the query
- [How to view the query cache](#)
- [How to build indices for a class](#) if you add an index after creating records
- [How to run the Tune Table facility](#)
- [How to move code from one database to another](#)
- [How to move data from one database to another](#)
- [How to copy a class](#)
- [Other sources of information on these topics](#)

12.1 Finding Definitions of Code Elements

This section describes how to find the actual code that defines code elements of different types. It discusses:

- [How to find a class member in Studio](#)
- [How to find a class member in the InterSystems Class Reference](#)
- [How to find subclasses of a class in the InterSystems Class Reference](#)
- [How to find a macro in Studio](#)

Also see the appendix “[What’s That?](#)”

12.1.1 Finding a Class Member in Studio

For a class member (property, method, parameter, and so on) to see its definition, you must find the class in which it is defined. To find the class quickly in Studio:

1. Press **Ctrl+Shift+f**
2. For **Find what**, type the member name.
3. For **File types**, choose `.cls`
4. Select **Find**.

Usually this returns a small number of classes to look at. You can determine which is the appropriate class by closer examination.

12.1.2 Finding a Class Member in the InterSystems Class Reference

To find a class member in the InterSystems Class Reference:

1. Select **Search**.
2. For **Search for**, select **All classes that define**.
3. For **Type**, select **Method** or other kind of class member.
4. For **Name**, type the name of the class member to find. The search is case-sensitive.
5. Select **Search**.

Usually this returns a small number of classes to look at. You can determine which is the appropriate class by closer examination.

12.1.3 Finding Subclasses in the InterSystems Class Reference

Sometimes you need to find all the subclasses of a certain class (for example, if you are rearranging the inheritances among your classes). The quickest way to do so is to open the class in the InterSystems Class Reference. The summary area displays a list of all the subclasses of your class.

12.1.4 Finding a Macro in Studio

The macro could be defined within the code you are currently looking at or it could be defined in an INC file that your code uses (including the system include files provided by Caché). An INC file can include other INC files, and a class inherits all the INC files used by any superclass.

To find the INC file that defines a macro, right-click the macro and select **Goto** *macroname*.

If the macro is not defined in an include file, do the following in Studio:

1. Press **Ctrl+Shift+f**
2. For **Find what**, type the macro name, without the preceding \$\$\$

Some of the macros supplied by Cache are documented in “[System-supplied Macro Reference](#)” in *Using Caché ObjectScript*.

12.2 Defining Databases

To create a local database:

1. Access the Management Portal.
2. Select **System Administration > Configuration > System Configurations > Local Databases**.
3. Select **Create New Database** to open the **Database Wizard**.
4. Enter the following information for the new database:
 - Enter a database name in the text box. Usually this is a short string containing alphanumeric characters; for rules, see “[Configuring Databases](#)” in the *Caché System Administration Guide*.
 - Enter a directory name or select **Browse** to select a database directory. If this is the first database you are creating, you must browse to the parent directory in which you want to create the database; if you created other databases, the default database directory is the parent directory of the last database you created.
5. Select **Finish**.

For additional options and information on creating remote databases, see “[Configuring Cache](#)” in the *Caché System Administration Guide*.

12.3 Defining Namespaces

To create a namespace that uses local databases:

1. Access the Management Portal.
2. Select **System Administration > Configuration > System Configurations > Namespaces**.
3. Select **Create New Namespace**.
4. Enter a **Name for the namespace**. Usually this is a short string containing alphanumeric characters; for rules, see “[Configuring Namespaces](#)” in the *Caché System Administration Guide*.
5. For **Select an existing database for Globals**, select a database or select **Create New Database**.

If you select **Create New Database**, the system prompts you with similar options as given in the [previous topic](#).

6. For **Select an existing database for Routines**, select a database or select **Create New Database**.

If you select **Create New Database**, the system prompts you with similar options as given in the [previous topic](#).

7. Select **Save**.

For additional options, see “[Configuring Cache](#)” in the *Caché System Administration Guide*.

12.4 Mapping a Global

When you *map a global to database ABC*, you configure a given namespace so that Caché writes this global to and reads this global from the database ABC, which is not the default database for your namespace. When you define this *global mapping*, Caché does not move the global (if it already exists) to the designated database; instead the mapping instructs Caché where to read and write the global in the future.

To map a global:

1. If the global already exists, move it to the desired database. See “[Moving Data from One Database to Another](#),” later in this chapter.
2. Access the Management Portal.
3. Select **System Administration > Configuration > System Configurations > Namespaces**.
4. Select **Global Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Global Mapping**.
6. For **Global database location**, select the database that should store this global.
7. Enter the **Global name** (omitting the initial caret from the name). You can use the * character to choose multiple globals.

The global does not have to exist when you map it (that is, it can be the name of a global you plan to create).

Note: Typically you create mappings for data globals for persistent classes, because you want to store that data in non-default databases. Often you can guess the name of the data globals, but remember that Caché automatically uses a hashed form of the class name if the name is too long. It is worthwhile to check the storage definitions for those classes to make sure you have the exact names of the globals that they use. See “[Storage](#),” earlier in this book.

8. Select **OK**.
9. To save the mappings, select **Save Changes**.

For more information, see the *Caché System Administration Guide*.

You can also define global mappings programmatically; see the “Globals” entry in the *InterSystems Programming Tools Index*.

The following shows an example global mapping, as seen in the Management Portal, which does not display the initial caret of global names:

The global mappings for namespace NOTES are displayed below:

Filter: <input type="text"/>	Page size: <input type="text" value="0"/>	Max rows: <input type="text" value="1000"/>	Results: 1	Page: < « 1 » > of 1
Global	Subscript	Database		
MyMappedGlobal		CACHETEMP	Edit	Delete

This mapping means the following:

- Within the namespace DEMONAMESPACE, if you set values of nodes of the global ^MyTempGlobal, you are writing data to the CACHETEMP database.

This is true whether you set the nodes directly or indirectly (via object access or SQL).

- Within the namespace DEMONAMESPACE, if you retrieve values from the global ^MyTempGlobal, you are reading data from the CACHETEMP database.

This is true whether you retrieve the values nodes directly or indirectly (via object access or SQL).

12.5 Mapping a Routine

When you *map a routine to database ABC*, you configure a given namespace so that Caché finds this routine in the database ABC, which is not the default database for your namespace. When you define this *routine mapping*, Caché does not move the routine (if it already exists) to the designated database; instead the mapping instructs Caché where to find the routine in the future.

To map a routine:

1. If the routine already exists, move it to the desired database as described in “[Moving Code from One Database to Another](#),” later in this chapter.
2. Access the Management Portal.
3. Select **System Administration > Configuration > System Configurations > Namespaces**.
4. Select **Routine Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Routine Mapping**.
6. For **Routine database location**, select the database that should store this routine.
7. Enter a value for **Routine name**. You can use the * character to choose multiple routines.

Use the actual routine name; that is, do not include a caret (^) at the start.

The routine does not have to exist when you map it (that is, it can be the name of a routine you plan to create).

8. Select the **Routine type**.
9. Select **OK**.
10. Select **OK**.
11. To save the mappings, select **Save Changes**.

For more information, see the [Caché System Administration Guide](#).

You can also define this kind of mapping programmatically. You can also define routine mappings programmatically; see the “Routines” entry in the *InterSystems Programming Tools Index*.

Important: When you map one or more routines, be sure to identify all the code and data needed by those routines, and ensure that all that code and data is available in all the target namespaces. The mapped routines could depend on the following items:

- Include files
- Other routines
- Classes
- Tables
- Globals

Use additional routine, package, and global mappings as needed to ensure that these items are available in the target namespaces.

12.6 Mapping a Package

When you *map a package to database ABC*, you configure a given namespace so that Caché finds the class definitions of this package in the database ABC, which is not the default database for your namespace. The mapping also applies to the generated routines associated with the class definitions; those routines are in the same package. This mapping does not affect the location of any stored data for persistent classes in these packages.

Also, when you define this *package mapping*, Caché does not move the package (if it already exists) to the designated database; instead the mapping instructs Caché where to find the package in the future.

To map a package:

1. If the package already exists, move the package to the desired database, as described in “[Moving Code from One Database to Another](#),” later in this chapter.
2. Access the Management Portal.
3. Select **System Administration > Configuration > System Configurations > Namespaces**.
4. Select **Package Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Package Mapping**.
6. For **Package database location**, select the database that should store this package.
7. Enter a value for **Package name**.

The package does not have to exist when you map it (that is, it can be the name of a package you plan to create).

8. Select **OK**.
9. Select **OK**.
10. To save the mappings, select **Save Changes**.

For more information, see the [Caché System Administration Guide](#).

You can also define this kind of mapping programmatically. You can also define package mappings programmatically; see the “Packages” entry in the *InterSystems Programming Tools Index*.

Important: When you map a package, be sure to identify all the code and data needed by the classes in that package, and ensure that all that code and data is available in all the target namespaces. The mapped classes could depend on the following items:

- Include files
- Routines
- Other classes
- Tables
- Globals

Use additional routine, package, and global mappings as needed to ensure that these items are available in the target namespaces.

12.7 Generating Test Data

Caché includes a utility for creating pseudo-random test data for persistent classes. The creation of such data is known as *data population*, and the utility for doing this is known as the Caché *populate utility*. This utility is especially helpful when testing how various parts of an application will function when working against a large set of data.

The populate utility consists of two classes: %Library.Populate and %Library.PopulateUtils. These classes provide methods that generate data of different typical forms. For example, one method generates random names:

ObjectScript

```
Write ##class(%Library.PopulateUtils).Name()
```

You can use the populate utility in two different ways.

12.7.1 Extending %Populate

In this approach, you do the following:

1. Add %Populate to the superclass list of your class.
2. Optionally specify a value for the *POPSPEC* parameter of each property in the class.

For the value of the parameter, specify a method that returns a value suitable for use as a property value.

For example:

Class Member

```
Property SSN As %String(POPSPEC = "##class(MyApp.Utils).MakeSSN()");
```

3. Write a utility method or routine that generates the data in the appropriate order: independent classes before dependent classes.

In this code, to populate a class, execute the **Populate()** method of that class, which it inherits from the %Populate superclass.

This method generates instances of your class and saves them by calling the **%Save()** method, which ensures that each property is validated before saving.

For each property, this method generates a value as follows:

- a. If the *POPSPEC* parameter is specified for that property, the system invokes that method and uses the value that it returns.
- b. Otherwise, if the property name is a name such as `City`, `State`, `Name`, or other predefined values, the system invokes a suitable method for the value. These values are hardcoded.
- c. Otherwise, the system generates a random string.

For details on how the `%Populate` class handles serial properties, collections, and so on, see “[The Caché Populate Utility](#)” in *Using Caché Objects*.

4. Invoke your utility method from the Terminal or possibly from any applicable startup code.

This is the general approach used for `Sample.Person` in the `SAMPLES` database.

12.7.2 Using Methods of `%Populate` and `%PopulateUtils`

The `%Populate` and `%PopulateUtils` classes provide methods that generate values of specific forms. You can invoke these methods directly, in the following alternative approach to data population:

1. Write a utility method that generates the data in the appropriate order: independent classes before dependent classes.

In this code, for each class, iterate a desired number of times. In each iteration:

- a. Create a new object.
- b. Set each property using a suitable random (or nearly random) value.
To do so, use a method of `%Populate` or `%PopulateUtils` or use your own method.
- c. Save the object.

2. Invoke your utility method from the Terminal.

This is the approach used for the two `DeepSee` samples in the `SAMPLES` database, contained in the `DeepSee` and `HoleFoods` packages.

12.8 Removing Stored Data

During the development process, it may be necessary to delete all existing test data for a class and then regenerate it (for example, if you have deleted the storage definition).

Here are two quick ways to delete stored data for a class (additional techniques are possible):

- Call the following class method:

```
##class(%ExtentMgr.Util).DeleteExtent(classname)
```

Where *classname* is the full package and class name.

- Delete the globals in which the data for the class and the indices for the class are stored. You may be more comfortable doing this through the Management Portal:
 1. Select **System Explorer > Globals**.
 2. Select **Delete**.
 3. On the left, select the namespace in which you are working.

4. On the right, select the check box next to the data global and the index global.
5. Select **Delete**.

The system prompts to confirm that you want to delete these globals.

These options delete the data, but not the class definition. (Conversely, if you delete the class definition, that does not delete the data.)

12.9 Resetting Storage

Important: It is important to be able to reset storage during development, but you never do this on a live system.

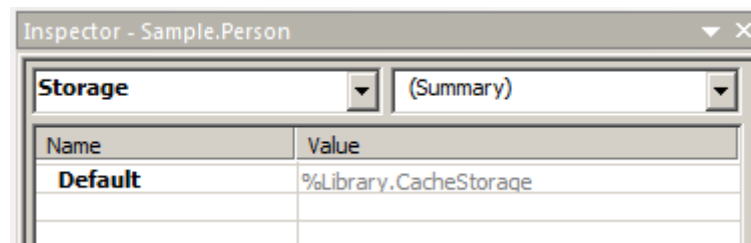
The action of resetting storage for a class changes the way that the class accesses its stored data. If you have stored data for the class, and if you have removed, added, or changed property definitions, and you then reset storage, you might not be able to access the stored data correctly. So if you reset storage, you should *also* delete all existing data for the class and regenerate or reload it, as appropriate.

To reset storage for a class in Studio:

1. In Studio, display the class.
2. Scroll to the end of the class definition.
3. Select the entire storage definition, starting with `<Storage name=` and ending with `</Storage>`. Delete the selection.
4. Save and recompile the class.

Or do this:

1. In Studio, display the class.
2. If the Inspector is not currently displayed, select **View > Inspector**.
3. In the upper left box of the Inspector, select **Storage**. Then Studio displays any defined storage definitions:



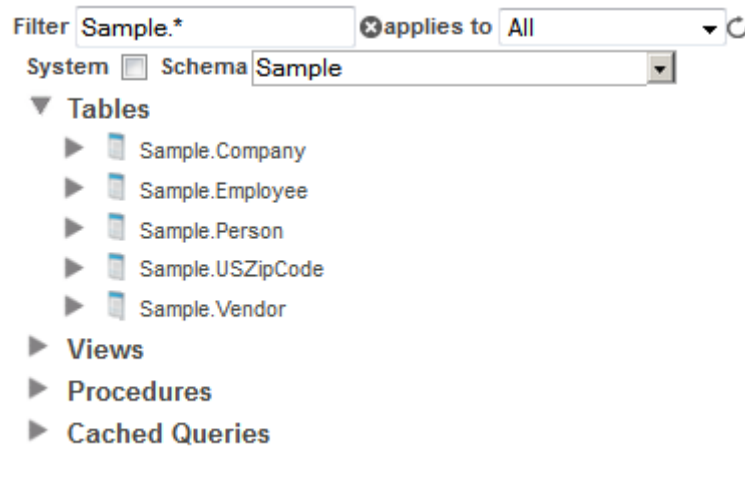
Typically there is only one storage definition, named **Default** as shown here.

4. Right-click the line for the storage definition and then select **Delete**.
5. Select **OK** to confirm.
6. Save and recompile the class.

12.10 Browsing a Table

To browse a table, do the following in the Management Portal:

1. Select **System Explorer** > **SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.
3. Optionally select an SQL schema from the **Schema** drop-down list. This list includes all SQL schemas in this namespace. Each schema corresponds to a top-level class package.
4. Expand the **Tables** folder to see all the tables in this schema. For example:



5. Select the name of the table. The right area then displays information about the table.
6. Select **Open Table**.

The system then displays the first 100 rows of this table. For example:

[Refresh](#) [Close Window](#)

Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First Street
2	2	17	05/30/1997	Green	Waterman,Danielle C.	944-39-5991		Oak Creek	ID	1648 Maple Street
3	3	86	04/01/1928		DeSantis,Christen N.	336-13-6311		Boston	AZ	8572 Maple Street
4	4	55	02/29/1960	Purple	Baker,Marvin Z.	198-22-7709		Queensbury	NV	1243 First Blvd
5	5	80	12/13/1934	Black	Diavolo,Ralph A.	586-13-9662		Hialeah	NY	3880 Maple Pl
6	6	38	10/13/1976		Russell,Paul S.	572-40-8824		Denver	CA	7269 Main Pl
7	7	33	11/26/1981	Purple Purple	Pascal,John X.	468-82-7179		Zanesville	AR	872 Elm Street

Note the following points:

- The values shown here are the display values, not the logical values as stored on disk.
- The first column (#) is the row number in the display.
- The second column (ID) is the unique identifier for a row in this table; this is the identifier to use when opening objects of this class. (In this class, these identifiers are integers, but that is not always true.)

These numbers happen to be the same in this case because this table is freshly populated each time the SAMPLES database is built. In a real application, it is possible that some records have been deleted, so that there are gaps in the **ID** values and the numbers here do not match the row numbers.

12.11 Executing an SQL Query

To run an SQL query, do the following in the Management Portal:

1. Select **System Explorer > SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.
3. Select **Execute Query**.
4. Type an SQL query into the input box. For example:

```
select * from sample.person
```

5. For the drop-down list, select **Display Mode**, **Logical Mode**, or **ODBC Mode**.

This controls how the user interface displays the results.

6. Then select **Execute**. Then the Portal displays the results. For example:

Row count: 200 Performance: 0.015 seconds 3442 global references Cached Query: [%sqlcq.SAMPLES.cls21](#) Last update: 2

ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Stree
1	34	50813		Ott,Liza F.	126-19-4431		Islip	OH	7433 Second Court
2	69	37939		Ingrahm,Sally N.	896-94-8820		Islip	IA	6707 Franklin Court
3	40	48586	OrangeGreen	Eagleman,Angela N.	937-68-7407		Denver	AL	4440 Madison Court
4	23	54736	Yellow	Ingersol,Umberto S.	381-48-8952		St Louis	WV	9908 Oak Blvd
5	11	59217		Mara,George F.	956-42-9085		Elmhurst	NE	5290 Ash Drive

12.12 Examining Object Properties

Sometimes the easiest way to see the value of a particular property is to open the object and write the property in the Terminal:

1. If the Terminal prompt is not the name of the namespace you want, then type the following and press return:

```
ZN "namespace"
```

Where *namespace* is the desired namespace.

2. Enter a command like the following to open an instance of this class:

```
set object=##class(package.class).%OpenId(ID)
```

Where *package.class* is the package and class, and *ID* is the ID of a stored object in the class.

3. Display the value of a property as follows:

```
write object.propname
```

Where *propname* is the property whose value you want to see.

12.13 Viewing Globals

To view globals in general, you can use the ObjectScript [ZWRITE](#) command or the **Globals** page in the Management Portal. If you are looking for the global that stores the data for a class, it is useful to first check the class definition to make sure you know the global to view.

1. If you are looking for the data global for a specific class and you are not sure which global stores the data for the class:
 - a. In Studio, display the class.
 - b. Scroll to the end of the class definition.
 - c. Find the `<DefaultData>` element. The value between `<DefaultData>` and `</DefaultData>` is the name of the global that stores data for this class.

Caché uses a simple naming convention to determine the names of these globals; see “[Globals Used by a Persistent Class](#),” earlier in this book. However, global names are limited to 31 characters (excluding the initial caret), so if the complete class name is long, the system automatically uses a hashed form of the class name instead.

2. In the Management Portal, select **System Explorer > Globals**.
3. If needed, select **Switch** in the header area to select the namespace in which you are interested.

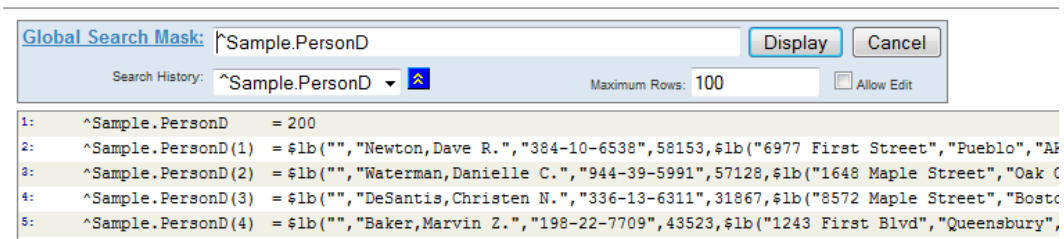
The Portal displays a list of the globals available in this namespace (notice that this display omits the initial caret of each name). For example:

Page size: 0		Results: 84	Page: < « 1 » » of 1
<input type="checkbox"/> Name	Location	Keep	Collation
<input type="checkbox"/> Aviation.AircraftD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.AircraftI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.Countries	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.CrewI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.EventD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.EventI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Aviation.States	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> CacheMsg	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit
<input type="checkbox"/> Cinema.ReviewD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard View Edit

Usually most non-system globals store data for persistent classes, which means that unless you display system globals, most globals will have familiar names.

4. Select **View** in the row for the global in which you are interested.

The system then displays the first 100 nodes of this global. For example:



5. To restrict the display to the object in which you are interested, append (*ID*) to the end of the global name in the **Global Search Mask** field, using the ID of the object. For example:

```
^Sample.PersonD(45)
```

Then press **Display**.

As noted earlier, you can also use the [ZWRITE](#) command, which you can abbreviate to ZW. Enter a command like the following in the Terminal:

```
zw ^Sample.PersonD(45)
```

12.14 Displaying INT Code

The system generates INT code for each routine and each class you compile.

To see the INT code for a given routine or class:

1. Make sure that the compiler keeps the INT code rather than discarding it:
 - a. Go to **Tools > Options**.
 - b. Select the **Compiler > General Flags** tab.
 - c. Select the **Keep Generated Source Code** option.

Also see “[System Qualifiers and Flags](#),” earlier in this book.

2. Display the routine or class in Studio.
3. If you changed the **Keep Generated Source Code** option, recompile the routine or class.
4. Select **View > View Other Code**.

If there are multiple generated routines, Studio displays a dialog box where you choose the routine to view. Otherwise it displays the generated routine.

The generated routine is displayed with a gray background to remind you that you should not edit this code. (The class compiler overwrites it the next time you compile the class.) The following shows an example:

```

;Sample.Person.1
;(C)InterSystems, generated for class Sample.Person.  Do NOT edit
;;C8091E66;Sample.Person
;
%BMEBuilt(bmeName)
    Set bmeName = "$Person"
    Quit '$d(^Sample.PersonI("$Person"))
%BindExport(%this,dev,Seen,RegisterOref,AllowedDepth,AllowedCapacity)
    i $d(Seen(+$this)) q 1
    Set Seen(+$this)=$this
    s sc = 1
    s proporef=..FavoriteColors
    s proporef=..Home
    s proporef=..Office
    s proporef=..Spouse
    d:RegisterOref InitObjVar^%SYS.BINDSRV($this)
    i dev'="" s t=$io u dev i $zobjexport($this_"",3)+$zobjexport(
If AllowedDepth>0 Set AllowedDepth = AllowedDepth - 1
If AllowedCapacity>0 Set AllowedCapacity = AllowedCapacity - 1,
s proporef=..FavoriteColors
    i proporef'="" ,dev'="" s t=$io u dev i $zobjexport(proporef,
if proporef'="" ,dev'="" d

```

Note: Some methods, routines, and subroutines are simple enough to be implemented in the kernel. There is no generated .INT code for such items.

Also see “[Documents \(“Files”\)](#),” earlier in this book.

12.15 Testing a Query and Viewing a Query Plan

In the Management Portal, you can test a query that your code will run. Here you can also view the query plan, which gives you information about how the Query Optimizer will execute the query. You can use this information to determine whether you should add indices to the classes or write the query in a different way.

To view a query plan, do the following in the Management Portal:

1. Select **System Explorer > SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.
3. Select **Execute Query**.
4. Type an SQL query into the input box. For example:

```
select * from sample.person
```

5. For the drop-down list, select **Display Mode**, **Logical Mode**, or **ODBC Mode**.

This controls how the user interface displays the results.

6. To test the query, select **Execute**.
7. To see the query plan, select **Show Plan**.

12.16 Viewing the Query Cache

For Caché SQL (except when used as embedded SQL), the system generates reusable code to access the data and places this code in the *query cache*. (For embedded SQL, the system generates reusable code as well, but this is contained within the generated INT code.)

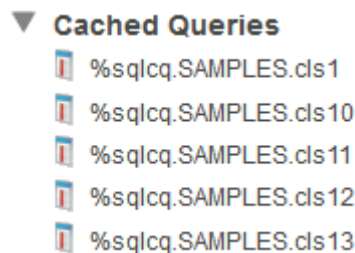
When you first execute an SQL statement, Caché optimizes the query and then generates and stores code that retrieves the data. It stores the code in the *query cache*, along with the optimized query text. Note that this cache is a cache of OBJ code, not of data.

Later when you execute an SQL statement, Caché optimizes it and then compares the text of that query to the items in the query cache. If Caché finds a stored query that matches the given one (apart from minor differences such as whitespace), it uses the code stored for that query.

The Management Portal groups the items in the query cache by schema. To view the query cache for a given schema, do the following in the Management Portal:

1. Select **System Explorer > SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.
3. Expand the **Cached Queries** folder.
4. Select the **Tables** link in the row for the schema.
5. At the top of the page, select **Cached Queries**.

The Portal displays something like this:



Each item in the list is OBJ code.

By default, Caché does not save the routine and INT code that it generates as a precursor to this OBJ code. You can force Caché to save this generated code as well. See “[Settings for Caché SQL](#),” earlier in this book.

You can purge cached queries (which forces Caché to regenerate this code). To purge cached queries, use **Actions > Purge Cached Queries**.

12.17 Building an Index

For Caché classes, indices do not require any maintenance, with one exception: if you add an index after you already have stored records for the class, you must build the index.

To do so:

1. Select **System Explorer > SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.

3. In the left area, select the table.
4. Select **Actions** > **Rebuild Indices**.

12.18 Using the Tune Table Facility

When the Query Optimizer decides the most efficient way to execute a specific SQL query, it considers, among other factors, the following items:

- How many records are in the tables
- For the columns used by the query, how nearly unique those columns are

This information is available only if you have run the Tune Table facility with the given table or tables. This facility calculates this data and stores it with the storage definition for the class, as the `<ExtentSize>` value for the class and the `<Selectivity>` values for the stored properties.

To use the Tune Table facility:

1. Select **System Explorer** > **SQL**.
2. If needed, select **Switch** in the header area to select the namespace in which you are interested.
3. In the left area, select the table.
4. Select **Actions** > **Tune Table**.

For `<Selectivity>` values, it is not necessary to do this again unless the data changes in character. For `<ExtentSize>`, it is not important to have an exact number. This value is used to compare the relative costs of scanning over different tables; the most important thing is to make sure that the relative values of *ExtentSize* between tables are correct (that is, small tables should have a small value and large tables a large one).

12.19 Moving Code from One Database to Another

If you need to move code from one database to another, you can do the following:

1. In Studio, go to a namespace that contains the code.
2. Create a new project and add all the code to it:
 - a. Select **File** > **New Project**.
 - b. Right-click each class, package, or routine that you want to move and select **Add to Project**.
 - c. Open the **Workspace** window and select the **Project** tab to make sure all the desired items are displayed.
3. Export the project:
 - a. Select **Tools** > **Export**.
 - b. Specify the file into which you wish to export the project. Either enter a file name (including its absolute or relative pathname) in the field or select **Browse** and navigate to the file.
 - c. Select **OK**.

This creates an `.xml` file. The format of the file is not documented, but it is fairly easy to read.

4. In Studio, import the XML file into the desired namespace:

- a. Switch to that namespace.
- b. Select **Tools > Import Local**.
- c. Navigate to and select the .xml file.
- d. Select **Open**.

Studio now displays a list of the code items in this file.

- e. Select each item that you want to import.
- f. Select **OK**.

5. Go back to the first namespace and delete the code.

12.20 Moving Data from One Database to Another

If you need to move data from one database to another, do the following:

1. Identify the globals that contain the data and its indices.

If you are not certain which globals a class uses, check its storage definition. See “[Storage](#),” earlier in this book.

2. Export those globals. To do so:

- a. In the Management Portal, select **System Explorer > Globals**.
- b. If needed, select **Switch** in the header area to select the namespace in which you are interested.
- c. Select the globals to export.
- d. Select **Export**.
- e. Specify the file into which you wish to export the globals. Either enter a file name (including its absolute or relative pathname) in the field or select **Browse** and navigate to the file.
- f. Select **Export**.

The globals are exported to a file whose extensions is .gof.

3. Import those globals into the other namespace. To do so:

- a. In the Management Portal, select **System Explorer > Globals**.
- b. If needed, select **Switch** in the header area to select the namespace in which you are interested.
- c. Select **Import**.
- d. Specify the import file. Either enter the file name or select **Browse** and navigate to the file.
- e. Select **Next** to view the contents of the file. The system displays a table of information about the globals in the specified file: the name of each global, whether or not it exists in the local namespace or database, and, if it does exist, when it was last modified.
- f. Choose those globals to import using the check boxes in the table.
- g. Select **Import**.

4. Go back to the first database and delete the globals, as described in “[Removing Stored Data](#).”

12.21 Renaming a Class

Studio does not provide a direct way to rename a class. To rename a class, you can take either of the following approaches:

- Create a copy of the class and delete the original.

To copy a class:

1. Select **Tools > Copy Class...**
2. For **Copy Class Definition From**, select the original class. This value is initialized with the name of the class that you are currently viewing.
3. For **To**, specify a new full class name.
4. Optionally select **Replace instances of Class Name**. This replaces instances of the class name within code in the copied class. It does not affect other classes, and it does not affect comments.
5. Select **OK**.

- Export the class from Studio, edit the XML file, reimport it, and then delete the original class.

To export and import, adapt the technique in “[Moving Code from One Database to Another](#).”

The XML structure of the exported file is not documented, but it is fairly easy to identify what must be changed. The class name is near the top of the file:

```
<Class name="MyPackage.MyClass">
```

The class definition might include additional references to the class name, which you would also change.

In either case, you should also do the following:

- If the class is a persistent class, delete or move any data stored with the original class. See “[Removing Stored Data](#).” Or adapt the technique described in “[Moving Data from one Database to Another](#).”
- Update references to the old class. Use **Ctrl+Shift+f** to find references to the old name.

12.22 For More Information

For more information on the topics covered in this chapter, see the following:

- *[Caché System Administration Guide](#)* describes how to use most of the Management Portal. This book includes information on configuring the Caché server, creating namespaces and databases, and creating mappings.
- *[Using Studio](#)* describes how to use Studio.
- *[Using the Terminal](#)* describes how to use the Terminal.
- *[Using Caché SQL](#)* includes more information on building indices, building indices programmatically, and using the Tune Table facility.

A

What's That?

As you read existing ObjectScript code, you may encounter unfamiliar syntax forms. This appendix shows syntax forms in different groups, and it explains what they are and where to find more information.

- [Non-alphanumeric characters within words](#)
- [Forms that start with .](#)
- [Forms that start with ..](#)
- [Forms that start with #](#)
- [Forms that start with \\$](#)
- [Forms that start with %](#)
- [Forms that start with ^](#)
- [Other forms](#)

This appendix does not list single characters that are obviously operators or that are obviously arguments to functions or commands.

Also see the following appendices in the *Caché ObjectScript Reference*.

- “[Symbols Used in ObjectScript](#)” lists all symbols.
- “[Abbreviations Used in ObjectScript](#)” lists the allowed short forms of commands and functions.

A.1 Non-Alphanumeric Characters in the Middle of “Words”

This section lists forms that look like words with non-alphanumeric characters in them. Many of these are obvious, because the operators are familiar. For example:

```
x>5
```

The less obvious forms are these:

```
abc^def
```

`def` is a routine, and `abc` is a label within that routine. `abc^def` is a subroutine.

Variation for `abc`:

- `%abc`

Some variations for `def`:

- `%def`
- `def.ghi`
- `%def.ghi`
- `def(xxx)`
- `%def(xxx)`
- `def.ghi(xxx)`
- `%def.ghi(xxx)`

`xxx` is an optional, comma-separated list of arguments.

A label can start with a percent sign but is purely alphanumeric after that.

A routine name can start with a percent sign and can include one or more periods. The caret is not part of its name. (In casual usage, however, it is very common to refer to a routine as if its name included an initial caret. Thus you may see comments about the `^def` routine. Usually you can tell from context whether the reference is to a global or to a routine.)

`i%abcdef`

This is an *instance variable*, which you can use to get or to set the value of the `abcdef` property of an object. See “[Object-specific ObjectScript Features](#)” in *Using Caché Objects*.

This syntax can be used only in an instance method. `abcdef` is a property in the same class or in a superclass.

For information on finding the property definitions, see “[Finding the Definition of a Class Member](#),” earlier in this book.

`abc->def`

Variations:

- `abc->def->ghi` and so on

This syntax is possible only within Caché SQL statements. It is an example of Caché *arrow syntax* and it specifies an implicit left outer join. `abc` is an object-valued field in the class that you are querying, and `def` is a field in the child class.

`abc->def` is analogous to Caché dot syntax (`abc.def`), which you cannot use in Caché SQL.

For information on Caché arrow syntax, see “[Implicit Joins \(Arrow Syntax\)](#)” in *Using Caché SQL*.

`abc?def`

Variation:

- `"abc"?def`

A question mark is the pattern match operator. In the first form, this expression tests whether the value in the variable `abc` matches the pattern specified in `def`. In the second form, `"abc"` is a string literal that is being tested. See “[Operators and Expressions](#)” in *Using Caché ObjectScript*.

Note that both the string literal `"abc"` and the argument `def` can include characters other than letters.

`"abc" ["def"`

Variations:

- `abc [def`
- `abc ["def "`
- `"abc" [def`

A left bracket (`[`) is the binary contains operator. In the first form, this expression tests whether the string literal `"abc"` contains the string literal `"def"`. In later forms, `abc` and `def` are variables that are being tested. See [“Operators and Expressions”](#) in *Using Caché ObjectScript*.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

`"abc"] "def"`

Variations:

- `abc] def`
- `abc] "def "`
- `"abc"] def`

A right bracket (`]`) is the binary follows operator. In the first form, this expression tests whether the string literal `"abc"` comes after the string literal `"def"`, in ASCII collating sequence. In later forms, `abc` and `def` are variables that are being tested. See [“Operators and Expressions”](#) in *Using Caché ObjectScript*.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

`"abc"]] "def"`

Variations:

- `abc]] def`
- `abc]] "def "`
- `"abc"]] def`

Two right brackets together (`]]`) are the binary sorts after operator. In the first form, this expression tests whether the string literal `"abc"` sorts after the string literal `"def"`, in numeric subscript collation sequence. In later forms, `abc` and `def` are variables that are being tested. See [“Operators and Expressions”](#) in *Using Caché ObjectScript*.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

A.2 . (One Period)

period within an argument list

Variations:

- `abc.def (.ghi)`
- `abc (.xyz)`

When you call a method or routine, you can pass an argument by reference or as output. To do so, place a period before the argument.

period at the start of a line

An older form of the Do command uses a period prefix to group lines of code together into a code block. See “[DO \(legacy version\)](#)” in the *Caché ObjectScript Reference*.

Code that uses this older form is sometimes referred to as *dotty*.

A.3 .. (Two Periods)

In every case, two periods together are the start of a reference from within a class member to another class member.

..abcdef

This syntax can be used only in an instance method (not in routines or class methods). `abcdef` is a property in the same class.

..abcdef (xxx)

This syntax can be used only in a method (not in routines). `abcdef ()` is another method in the same class, and `xxx` is an optional comma-separated list of arguments.

..#abcdef

This syntax can be used only in a method (not in routines). `abcdef` is a parameter in this class.

In classes provided by InterSystems, all parameters are defined in all capitals, by convention, but your code is not required to do this.

Remember that the pound sign is *not* part of the parameter name.

For information on finding the class member definition, see “[Finding the Definition of a Class Member](#),” earlier in this book.

A.4 # (Pound Sign)

This section lists forms that start with a pound sign.

#abcdef

In most cases, `#abcdef` is a preprocessor directive. Caché provides a set of preprocessor directives. Their names start with either one or two pound signs. Here are some common examples:

- **#define** defines a macro (possibly with arguments)
- **#deflargs** defines a macro that has one argument that includes commas
- **#sqlcompile mode** specifies the compilation mode for any subsequent embedded SQL statements

For reference information and other directives, see “[ObjectScript Macros and the Macro Preprocessor](#)” in *Using Caché ObjectScript*.

Less commonly, the form `#abcdef` is an argument used with specific commands (such as [READ](#) and [WRITE](#)), special variables, or routines. For details, consult the reference information for the command, variable, or routine that uses this argument.

##abcdef

`##abcdef` is a preprocessor directive. See the comments for `#abcdef`.

##class(abc.def).ghi(xxx)

Variation:

- `##class(def).ghi(xxx)`

`abc.def` is a package and class name, `ghi` is a class method in that class, and `xxx` is an optional comma-separated list of arguments.

If the package is omitted, the class `def` is in the same package as the class that contains this reference.

##super()

Variations:

- `##super(abcdef)`

This syntax can be used only in a method. It invokes the overridden method of the superclass, from within the current method of the same name in the current class. `abcdef` is a comma-separated list of arguments for the method. See “[Object-specific ObjectScript Features](#)” in *Using Caché Objects*.

A.5 Dollar Sign (\$)

This section lists forms that start with a dollar sign.

\$abcdef

Usually, `$abcdef` is a special variable. See “[ObjectScript Special Variables](#)” in the *Caché ObjectScript Reference*.

`$abcdef` could also be a custom special variable. See “[Extending ObjectScript with %ZLang](#)” in the chapter “Customizing Caché” in *Caché Specialized System Tools and Utilities*.

\$abcdef(xxx)

Usually, `$abcdef()` is a system function, and `xxx` is an optional comma-separated list of arguments. For reference information, see the *Caché ObjectScript Reference*.

`$abcdef()` could also be a custom function. See “[Extending ObjectScript with %ZLang](#)” in the chapter “Customizing Caché” in *Caché Specialized System Tools and Utilities*.

\$abc.def.ghi(xxx)

In this form, `$abc` is `$SYSTEM` (in any case), `def` is the name of class in the `%SYSTEM` package, `ghi` is the name of a method in that class, and `xxx` is an optional comma-separated list of arguments for that method.

The `$SYSTEM` special variable is an alias for the `%SYSTEM` package, to provide language-independent access to methods in classes of that package. For example: `$SYSTEM.SQL.DATEDIFF`

For information on the methods in this class, see the *InterSystems Class Reference*.

\$\$abc

Variation:

- `$$abc (xxx)`

`abc` is a subroutine defined within the routine or the method that contains this reference. This syntax invokes the subroutine `abc` and gets its return value. See the chapter “[User-defined Code](#)” in *Using Caché ObjectScript*.

\$\$abc^def

Variations:

- `$$abc^def (xxx)`
- `$$abc^def .ghi`
- `$$abc^def .ghi (xxx)`

This syntax invokes the subroutine `abc` and gets its return value. The part after the caret is the name of the routine that contains this subroutine. See the chapter “[User-defined Code](#)” in *Using Caché ObjectScript*.

\$\$\$abcdef

`abcdef` is a macro; note that the dollar signs are not part of its name (and are thus not seen in the macro definition).

Some of the macros supplied by Caché are documented in “[System-supplied Macro Reference](#)” in *Using Caché ObjectScript*. Otherwise, see “[Finding a Macro in Studio](#),” earlier in this book.

In casual usage, it is common to refer to a macro as if its name included the dollar signs. Thus you may see comments about the `$$$abcdef` macro.

A.6 Percent Sign (%)

By convention, most packages, classes, and methods in Caché system classes start with a percent character. From the context, it should be clear whether the element you are examining is one of these. Otherwise, the possibilities are as follows:

%abcdef

`%abcdef` is one of the following:

- A local variable, including possibly a local variable set by Caché.
- A routine.

Variation:

– `%abcdef .ghijkl`

- An embedded SQL variable (these are `%msg`, `%ok`, `%ROWCOUNT`, and `%ROWID`).

For information, see the section “[System Variables](#)” in the chapter “Using Embedded SQL” in *Using Caché SQL*.

- A Caché SQL command, function, or predicate condition (for example, `%STARTSWITH` and `%SQLUPPER`).

Variation:

– `%abcdef (xxx)`

For information, see the [Caché SQL Reference](#).

%%abcdef

%%abcdef is %%CLASSNAME, %%CLASSNAMEQ, %%ID, or %%TABLENAME. These are pseudo-field keywords. For details, see the [Caché SQL Reference](#).

A.7 Caret (^)

This section lists forms that start with a caret, from more common to less common.

^abcdef

Variation:

- ^%abcdef

There are three possibilities:

- ^abcdef or ^%abcdef is a global.
- ^abcdef or ^%abcdef is an argument of the **LOCK** command. In this case, ^abcdef or ^%abcdef is a lock name and is held in the lock table (in memory).
- abcdef or %abcdef is a routine. The caret is not part of the name, but rather part of the syntax to call the routine.

In casual usage, it is very common to refer to a routine as if its name included an initial caret. Thus you may see comments about the ^abcdef routine. Usually you can tell from context whether the reference is to a global or to a routine. Lock names appear only after the **LOCK** command; they cannot be used in any other context.

^\$abcdef

Variation:

- ^\$|"ghijkl"|abcdef

Each of these is a structured system variable, which provides information about globals, jobs, locks, or routines.

\$abcdef is \$GLOBAL, \$JOB, \$LOCK, or \$ROUTINE.

ghijkl is a namespace name.

Caché stores information in the following system variables:

- [^\\$GLOBAL](#)
- [^\\$JOB](#)
- [^\\$LOCK](#)
- [^\\$ROUTINE](#)

See the *Caché ObjectScript Reference*.

^||abcdef

Variations:

- `^| "^"| abcdef`
- `^["^"] abcdef`
- `^["^" , " "] abcdef`

Each of these is a *process-private global*, a mechanism for temporary storage of large data values. Caché uses some internally but does not present any for public use. You can define and use your own process-private globals. See “[Variables](#)” in *Using Caché ObjectScript*.

`^|xxx|abcdef`

Some variations:

- `^|xxx|%abcdef`
- `^[xxx]abcdef`
- `^[xxx]%abcdef`

Each of these is an *extended reference* — a reference to a global or a routine in another namespace. The possibilities are as follows:

- `^abcdef` or `^%abcdef` is a global in the other namespace.
- `abcdef` or `%abcdef` is a routine in the other namespace.

Extended references were sometimes necessary before Caché provided support for global and routine mappings.

The `xxx` component indicates the namespace. This is either a quoted string or an unquoted string. See “Extended References” in the chapter “[Syntax Rules](#)” in *Using Caché ObjectScript*.

`^abc^def`

This is an implied namespace. See the “[ZNSPACE](#)” entry in the *Caché ObjectScript Reference*.

`^^abcdef`

This is an implied namespace. See the “[ZNSPACE](#)” entry in the *Caché ObjectScript Reference*.

A.8 Other Forms

`+abcdef`

Some variations:

- `+^abcdef`
- `+"abcdef"`

Each of these expressions returns a number. In the first version, `abcdef` is the name of a local variable. If the contents of this variable do not start with a numeric character, the expression returns 0. If the contents do start with a numeric character, the expression returns that numeric character and all numeric characters after it, until the first nonnumeric character. For a demonstration, run the following example:

ObjectScript

```
write +"123abc456"
```

See “[String Relational Operators](#)” in the chapter “Operators and Expressions” in *Using Caché ObjectScript*.

`{"abc":(def),"abc":(def),"abc":(def)}`

This syntax is a [JSON object literal](#) and it returns an instance of %DynamicObject. "abc" is the name of a property, and def is the value of the property. For details, see [Using JSON in Caché](#).

`{abcdef}`

This syntax is possible where Caché SQL uses ObjectScript. abcdef is the name of a field. See “[Referring to Fields from ObjectScript](#)” in *Using Caché Objects*.

`{%%CLASSNAME}`

This syntax can be used within trigger code and is replaced at class compilation time.

Others:

- `{%%CLASSNAMEQ}`
- `{%%ID}`
- `{%%TABLENAME}`

These items are not case-sensitive. See “[CREATE TRIGGER](#)” in the *Caché SQL Reference*.

`&sql(xxx)`

This is embedded SQL and can be used anywhere ObjectScript is used. xxx is one SQL statement. See “[Using Embedded SQL](#)” in *Using Caché SQL*.

`&js<xxx>`

This syntax is possible in Zen pages and it indicates embedded JavaScript commands. xxx is one or more JavaScript commands. See “[Zen Pages](#)” in *Developing Zen Applications*.

`&html<xxx>`

This syntax is possible in Zen pages and it indicates embedded HTML commands. xxx is one or more HTML commands. See “[Zen Pages](#)” in *Developing Zen Applications*.

`[abcdef,abcdef,abcdef]`

This syntax is a [JSON array literal](#) and it returns an instance of %DynamicArray. abcdef is an item in the array. For details, see *Using JSON in Caché*.

`*abcdef`

Special syntax used by the following functions and commands:

- [\\$ZSEARCH](#)
- [\\$EXTRACT](#)
- [WRITE](#)
- [\\$ZTRAP](#)
- [\\$ZERROR](#)

See these items in the *Caché ObjectScript Reference*.

?abcdef

The question mark is the pattern match operator and `abcdef` is the comparison pattern. See “[Operators and Expressions](#)” in Using Caché ObjectScript.

@abcdef

The at sign is the indirection operator. See “[Operators and Expressions](#)” in Using Caché ObjectScript.

B

Rules and Guidelines for Identifiers

For convenience, this appendix summarizes the rules for ObjectScript identifiers in all server-side contexts and provides guidelines to avoid name collisions.

- [Namespaces](#)
- [Databases](#)
- [Local variables](#)
- [Global variables](#)
- [Routines and labels](#)
- [Classes](#)
- [Class members](#)
- [Custom items in CACHESYS](#)

Also note that ObjectScript does not have reserved words, so if you use a command as an identifier, the result is syntactically correct, but the code is also potentially confusing to anyone who reads it.

For identifiers for security entities such as users, roles, and resources, see the relevant section of the [Caché Security Administration Guide](#).

B.1 Namespaces

In a namespace name, the first character must be a letter or a percent sign (%). The remaining characters must be letters, numbers, hyphens, or underscores. The name cannot be longer than 255 characters.

B.1.1 Namespace Names to Avoid

The following namespace names are reserved: %SYS, BIN, BROKER, DOCBOOK and DOCUMATIC. Some of these namespaces are described in “[System-supplied Namespaces](#),” earlier in this book. Others are used internally by Caché.

Also, if you use the namespace name SAMPLES, you must take care not to install the InterSystems code samples database, which uses that namespace.

B.2 Databases

In a database name, the first character must be a letter or an underscore. The remaining characters must be letters, numbers, hyphens, or underscores. The name cannot be longer than 30 characters.

B.2.1 Database Names to Avoid

The following database names are reserved: CACHE, CACHESYS, CACHELIB, CACHEAUDIT, CACHETEMP, and DOCBOOK.

Also, if you use the database name SAMPLES, you must take care not to install the InterSystems code samples database, which has that name.

For information on these databases, see “[System-supplied Databases](#),” earlier in this book.

B.3 Local Variables

For the name of local variable, the following rules apply in ObjectScript:

- The first character must be either a letter or a percent sign (%).
If you start a name with %, use z or Z as the next character after that.
- The remaining characters must be letters or numbers. On Unicode systems, these other characters can include letter characters above ASCII 255 (Unicode letters).
- Names are case-sensitive.
- The name must be unique (within the appropriate context) to the first 31 characters.
Any subscripts of the variable do not contribute to this count.

B.3.1 Local Variable Names to Avoid

Avoid using the following names for local variables:

- `SQLCODE`
Avoid using `SQLCODE` as the name of a variable in any context where Caché SQL might run. See the “[SQLCODE Values and Error Messages](#)” chapter of the *Caché Error Reference*.
- `zenPage`, `this`, and `zenThis`
In Zen classes, avoid using these as names of variables. See the “[Zen Special Variables](#)” section in the “Zen Pages” chapter in *Developing Zen Applications*.

B.4 Global Variables

For the name of a global variable, the following rules apply in ObjectScript:

- The first character must be a caret (^), and the next character must be either a letter or a percent sign (%). For global names, a letter is defined as being an alphabetic character within the range of ASCII 65 through ASCII 255.

- The remaining characters must be letters or numbers (with one exception, noted in the next bullet).
- The name of a global variable can include one or more period (.) characters, except not as the first or last character.
- Names are case-sensitive.
- The name must be unique (within the appropriate context) to the first 31 characters. The caret character does not contribute to this count. That is, the name of a global variable must be unique to the first 32 characters, including the caret. Any subscripts of the variable do not contribute to this count.
- In the CACHESYS database, InterSystems reserves to itself all global names *except* those starting with ^z, ^Z, ^%z, and ^%Z. See “[Custom items in CACHESYS.](#)”

In all other databases, InterSystems reserves all global names starting with ^ISC. and ^%ISC.

Also see the [following subsection](#).

B.4.1 Global Variable Names to Avoid

When you create a database, Caché initializes it with some globals for its own use. Also, every namespace that you create contains mappings to system globals, including global nodes that are in writable system databases.

To avoid overwriting system globals, do not set, modify, or kill the following globals in any namespace:

- Globals with names that start with ^%, with the following exceptions:
 - Your own globals with names that start ^%z or ^%Z
 - ^%SYS (you can set nodes as noted in the documentation)
- ^BP (restriction applies only to a namespace in which you are using Caché MVBasic)
- ^COMO (restriction applies only to a namespace in which you are using Caché MVBasic)
- ^CacheAuditD
- ^CacheMsg, the message dictionary; see the article [String Localization and Message Dictionaries](#).
- ^CacheTemp and globals whose names start with ^CacheTemp, with one exception:
 - Your own globals with names that start with ^CacheTempUser
- ^CMQLlog (restriction applies only to a namespace in which you are using Caché MVBasic; you can set or kill this global as noted in the documentation)
- ^D.1 and ^D.2 (restriction applies only to a namespace in which you are using Caché MVBasic)
- ^DeepSee.* (restriction applies only to a namespace in which you are using DeepSee)
- ^DICT.* (restriction applies only to a namespace in which you are using Caché MVBasic)
- ^ERRORS
- ^InterSystems.Sequences (restriction applies only to a namespace in which you are using the Caché Hibernate dialect; see “Using the Caché Hibernate Dialect” in *Using Caché with JDBC*.)
- ^ISC.* (reserved for use by InterSystems)
- ^ISCDebugLevel (except for setting nodes as noted in the documentation)
- ^ISCMonitor (except for setting nodes as noted in the documentation)
- ^ISCSOAP (except for setting nodes as noted in the documentation)
- ^mqh (SQL query history)

- **^mtemp***
- **^MV.*** (restriction applies only to a namespace in which you are using Caché MVBasic)
- **^OBJ.GUID** (except as noted in the documentation)
- **^OBJ.DSTIME**
- **^OBJ.JournalT**
- **^oddBIND**
- **^oddCOM**
- **^oddDEF** (contains class definitions)
- **^oddDEP**
- **^oddEXT**
- **^oddEXTR**
- **^oddFMD** (restriction applies only to a namespace in which you are the FileMan Mapping Utility)
- **^oddMAP**
- **^oddMETA**
- **^oddPKG**
- **^oddPROC**
- **^oddPROJECT**
- **^oddSQL**
- **^oddStudioDocument**
- **^oddStudioMenu**
- **^oddTSQL**
- **^oddXML**
- **^PH** (restriction applies only to a namespace in which you are using Caché MVBasic)
- **^rBACKUP**
- **^rINC** (contains include files)
- **^rINCSAVE**
- **^rINDEX**
- **^rINDEXCLASS**
- **^rINDEXEXT**
- **^rINDEXSQL**
- **^rMAC** (contains MAC code)
- **^rMACSAVE**
- **^rMAP**
- **^rOBJ** (stores OBJ code)
- **^ROUTINE** (stores routines)
- **^SAVEDLISTS** (restriction applies only to a namespace in which you are using Caché MVBasic)

- **^SPOOL** (restriction applies only to a namespace in which you are using Caché spooling; see “[Spool Device](#)” in the *Caché I/O Device Guide*)
- **^SYS** (except for setting nodes as noted in the documentation)
- **^z*** and **^Z*** (reserved for use by InterSystems)

B.5 Routines and Labels

For the name of a routine or for a label, the following rules apply in ObjectScript:

- The first character must be either a letter or a percent sign (%).
If you start a routine name with %, use z or Z as the next character after that; see “[Custom items in CACHESYS.](#)”
- The remaining characters must be letters or numbers (with one exception; see the next bullet). On Unicode systems, these other characters may include any letter character above ASCII 128.
- The name of a routine can include one or more period (.) characters, except not as the first or last character.
- Names are case-sensitive.
- The name of a routine must be unique (within the appropriate context) within the first 255 characters.
A label must be unique (within the appropriate context) within the first 31 characters.

Note that certain %Z routine names are reserved for your use. See the subsection.

B.5.1 Reserved Routine Names for Your Use

Caché reserves the following routine names for your use. These routines do not exist, but if you define them, the system automatically calls them when specific events happen.

- The **^ZWELCOME** routine is intended to contain custom code to execute when the Terminal starts. See [Using the Terminal](#).
- The **^ZAUTHENTICATE** and **^ZAUTHORIZE** routines are intended to contain custom code for authentication and authorization (to support *delegated authentication* and *delegated authorization*). For these routines, Caché provides templates. See the [Caché Security Administration Guide](#).
- The **^ZMIRROR** routine is intended to contain code to customize failover behavior when you use Caché Mirroring. See the [Caché High Availability Guide](#).
- The **^%ZSTART** and **^%ZSTOP** routines are intended to contain custom code to execute when certain events happen, such as when a user logs in. These routines are not predefined. If you define them, the system can call them when these events happen. See “[Customizing Start and Stop Behavior with ^%ZSTART and ^%ZSTOP Routines](#)” in *Caché Specialized System Tools and Utilities*.
- **^%ZLANGV00** and other routines with names that start **^%ZLANG** are intended to contain your custom variables, commands, and functions. See “[Extending Languages with %ZLANG Routines](#)” in *Caché Specialized System Tools and Utilities*.
- The **^%ZJREAD** routine is intended to contain logic to manipulate journal files if you use the **^JCONVERT** routine. See the chapter “[Journaling](#)” in the *Caché Data Integrity Guide*.

B.6 Classes

For any class, the full class name has the following form: *packagename.classname*

The rules for class names are as follows:

- *packagename* (the package name) and *classname* (the short class name) must each start with a letter.
- *packagename* can include periods.

If so, the character immediately after any period must be a letter.

Each period-delimited piece of *packagename* is treated as a subpackage name and is subject to uniqueness rules.

- The remaining characters must be letters or numbers.

On Unicode systems, these other characters can include letter characters above ASCII 128.

- The package name and the short class name must be unique. Similarly, any subpackage name must be unique within the parent package name.

Note that the system preserves the case that you use when you define each class, and you must exactly match the case as given in the class definition. However, two identifiers cannot differ only in case. For example, the identifiers `id1` and `ID1` are considered identical for purposes of uniqueness.

- There are length limits:
 - The package name (including all periods) must be unique within the first 189 characters.
 - The short class name must be unique within the first 60 characters.

The full class name contributes to the individual length limits for class members; see the next section.

B.6.1 Class Names to Avoid

For persistent classes, avoid using an SQL reserved word as the short name for the class.

If you use an SQL reserved word as the short name for a class, you will need to specify the [SqlTableName](#) keyword for the class. Also, the mismatch between the short class name and the SQL table name will require greater care when reading the code in the future.

For a list of the SQL reserved words, see “[Reserved Words](#)” in the *Caché SQL Reference*.

B.7 Class Members

For a class member that you create, unless the name of that item is delimited, the name must follow these rules:

- The name must start with either a letter or a percent sign (%).

There is an additional consideration for class members that are projected to SQL (this includes, for example, most properties of persistent classes). If the first character is %, the second character must be Z or z.

- The remaining characters must be letters or numbers. On Unicode systems, these other characters can include letter characters above ASCII 128.
- The member name must be unique (within the appropriate context).

Note that the system preserves the case that you use when you define classes, and you must exactly match the case as given in the class definition. However, two class members cannot have names that differ only in case. For example, the identifiers `id1` and `ID1` are considered identical for purposes of uniqueness.

- A method or property name must be unique within the first 180 characters.
- The combined length of the name of a property and of any indices on the property should be no longer than 180 characters.
- The full name of each member (including the unqualified member name and the full class name) must be less than or equal to 220 characters.
- InterSystems strongly recommends that you do not give two members the same name. This can have unexpected results.

As of release 2012.2, member names can be delimited. To create a delimited member name, use double quotes for the first and last characters of the name. Then the name can include characters that are otherwise not permitted. For example:

Class Member

```
Property "My Property" As %String;
```

B.7.1 Member Names to Avoid

For persistent classes, avoid using an SQL reserved word as the name of a member.

If you use an SQL reserved word for one of these names, you will have to do extra work to specify how the class is projected to SQL. For example, for a property, you would need to specify the [SqlFieldName](#) keyword. Also, the mismatch between the identifier in the class and the identifier in SQL will require greater care when reading the code in the future.

For a list of the SQL reserved words, see “[Reserved Words](#)” in the *Caché SQL Reference*. Notice that this list includes many items with names beginning with %, such as `%UPPER` and `%CONTAINS`. Such items are InterSystems extensions to SQL, and additional extensions may be added in future releases.

B.8 Custom Items in CACHESYS

You can create items in the CACHESYS database. When you install a Caché upgrade, this database is upgraded. During this upgrade, some items are deleted unless they follow the naming conventions for custom items.

To add code or data to this database so that your items are not overwritten, do one of the following:

- Go to the `%SYS` namespace and create the item. For this namespace, the default routine database and default global database are both CACHESYS. Use the following naming conventions to prevent your items from being affected by the upgrade installation:
 - Classes: start the package with `Z` or `z`
 - Routines: start the name with `Z`, `z`, `%Z`, or `%z`
 - Globals: start the name with `^Z`, `^z`, `^%Z`, or `^%z`
- In any namespace, create items with the following names:
 - Routines: start the name with `%Z` or `%z`
 - Globals: start the name with `^%Z` or `^%z`

Because of the standard mappings in any namespace, these items are written to CACHESYS.

MAC code and include files are not affected by upgrade.

C

General System Limits

This appendix lists some of the limits that are applicable across all server-side languages. It discusses the following topics:

- [Long string limit](#)
- [Class-only limits](#)
- [Limits per class or routine](#)
- [Other programming limits](#)

For limits on identifier names, see “[Rules and Guidelines for Identifiers](#).”

For additional system-wide limits, see the [Caché Parameter File Reference](#).

C.1 Long String Limit

There is a limit to the length of a value of a variable. If you have *long strings* enabled in your installation (as they are by default in new installations as of release 2012.2), the limit is 3,641,144 characters. If long strings are not enabled, the limit is 32,767 characters.

To enable or disable long strings, you can use the Management Portal, as described in “[Enabling Long String Operations](#),” earlier in this book. (Or, in the Caché parameter file, specify the value of the *EnableLongStrings* parameter, as described in the [EnableLongStrings](#) section of the *Caché Parameter File Reference*.)

Caché also supports the use of long strings on an optional, per-instance basis. To enable long strings for the current instance, use the *EnableLongStrings* property of the *Config.Miscellaneous* class.

When a process actually uses a long string, the memory for the string comes from the operating system’s `malloc()` buffer, not from the partition memory space for the process. Thus the memory allocated for actual long string values is *not* subject to the limit set by the **Maximum per Process Memory** setting on the **System Memory and Startup Settings** page and does not affect the [\\$STORAGE](#) value for the process.

Important: It is important to realize that “strings” are not just the result of reading from input/output devices. They can show up in other contexts such as the data in the rows of a resultset returned by an SQL query, by construction of \$LISTs with a large number of items, as the output of an XSLT transformation, and many other ways.

If long strings are not enabled, Caché cannot guarantee that an application can take advantage of published limits. These limits all assume that long strings are enabled, unless otherwise noted.

C.2 Class Limits

The following limits apply only to classes:

class inheritance depth

Limit: 50. A given class can be subclassed to a depth of 50 but not further.

foreign keys

Limit: 400 per class.

indices

Limit: 400 per class.

methods

Limit: 2000 per class.

parameters

Limit: 1000 per class.

projections

Limit: 200 per class.

properties

Limit: 1000 per class.

queries

Limit: 200 per class.

SQL constraints

Limit: 200 per class.

storage definitions

Limit: 10 per class.

superclasses

Limit: 127 per class.

triggers

Limit: 200 per class.

XData blocks

Limit: 1000 per class.

C.3 Class and Routine Limits

The following limits apply to both classes and routines:

class method references

Limit: 32768 unique references per routine or class.

The following is counted as two class method references because the class name is different even though the method name is the same.

ObjectScript

```
Do ##class(c1).abc(), ##class(c2).abc()
```

class name references

Limit: 32768 unique references per routine or class.

For example, the following is counted as two class name references:

```
Do ##class(c1).abc(), ##class(c2).abc()
```

Similarly, the following is counted as two class references because the normalization of %File to %Library.File is done at runtime, not at compile time.

```
Do ##class(%File).Open(x)
Do ##class(%Library.File).Open(y)
```

instance method references

Limit: 32768 per routine or class.

If X and Y are OREFs, the following counts as one instance method reference:

```
Do X.abc(), Y.abc()
```

References to multidimensional properties are counted as instance methods because the compiler cannot distinguish between them. For example, consider the following statement:

```
Set var = OREF.xyz(3)
```

Because the compiler cannot tell whether this statement refers to the method **xyz()** or to the multi-dimensional property **xyz**, it counts this as an instance method reference.

lines

Limit: 65535 lines per routine, including comment lines. The limit applies to the size of the INT representation.

literals (ASCII)

Limit: 65535 ASCII literals per routine or class.

An ASCII literal is a quoted string of three or more characters where no character is larger than \$CHAR(255).

Note that ASCII literals and Unicode literals are handled separately and have separate limits.

literals (Unicode)

Limit: 65535 Unicode literals per routine or class.

A Unicode literal is a quoted string with at least one character larger than \$CHAR(255).

Note that ASCII literals and Unicode literals are handled separately and have separate limits.

parameters

Limit: 255 parameters per subroutine, method, or stored procedure.

procedures

Limit: 32767 per routine.

property read references

Limit: 32768 per routine or class.

This limit refers to reading the value of a property as in the following example:

ObjectScript

```
Set X = OREF.prop
```

property set references

Limit: 32768 per routine or class.

This limit refers to setting the value of a property as in the following example:

ObjectScript

```
Set OREF.prop = value
```

routine references

Limit: 65535 per routine or class.

This limit applies to the number of unique references (^routine) in a routine or class.

target references

Limit: 65535 per routine or class.

A target is label^routine (a combination of label and routine).

Any target reference also counts as a routine reference. For example, the following is counted as two routine references and three target references:

```
Do Label1^Rtn, Label2^Rtn, Label1^Rtn2
```

TRY blocks

Limit: 65535 per routine.

variables (private)

Limit (ObjectScript): 32763 per procedure.

Limit (Caché Basic or MVBasic): 32759 per routine.

variables (public)

Limit (ObjectScript): 65503 per routine or class.

Limit (Caché Basic or MVBasic): 65280 per routine or class.

For limits on the lengths of variable names and other identifiers, see “[Rules and Guidelines for Identifiers](#),” earlier in this book.

C.4 Other Programming Limits

The following table lists other limits that are relevant when writing code.

%Status value limits

Length limit of error message: Just under 32k characters.

Maximum number of %Status values that can be combined into a single %Status value: 150.

{ } nesting

Limit: 32767 levels.

This is the maximum depth of nesting of any language element that uses curly braces, like IF { FOR { WHILE { ... } } }.

characters per line

Limit: 65535 characters per line, if long strings are enabled. Note that in Studio, you cannot edit lines longer than 32767 characters.

global subscript, length

See “[Maximum Length of a Global Reference](#)” in *Using Caché Globals*.

global reference, length

Limit: 511 encoded characters (which may be fewer than 511 typed characters). The term *global reference* refers to the name of the global plus all of its subscripts. For a discussion, see “[Maximum Length of a Global Reference](#)” in *Using Caché Globals*.

numeric value

Limits (for decimal or native format): Approximately 1.0E-128 to 9.22E145. See “[Numeric Computing in InterSystems Applications](#),” later in this book.

Limits (for double format): See “[Numeric Computing in InterSystems Applications](#),” later in this book.

D

Numeric Computing in InterSystems Applications

This appendix provides details on the numeric formats supported by Caché. It discusses the following topics:

- [Representations of numbers in Caché](#)
- [Items to consider when you choose a numeric format](#)
- [How to convert numeric representations](#)
- [Operations that involve numbers](#)
- [Changes that occurred in release 2008.2](#)
- [Additional sources of information](#)

D.1 Representations of Numbers

Caché has two different ways of representing numbers.

- The first of these has its roots in the original implementation of Caché. This representation will be referred to as *decimal format*.

In class definitions, you use the `%Library.Decimal` datatype class when you want a property to contain a decimal format number.

- The second, more recently supported, form adheres to the IEEE Binary Floating-Point Arithmetic standard (#754–2019). This latter format is referred to as *\$DOUBLE format* after the ObjectScript function (`$DOUBLE`) that is used to convert numbers into this form.

In class definitions, you use the `%Library.Double` datatype class when you want a property to contain a \$DOUBLE format number.

D.1.1 Decimal Format

Caché represents decimal numbers internally in two parts. The first is called the *significand*, and the second is called the *exponent*:

- The significand contains the significant digits of the number. It is stored as a signed 64-bit integer with the decimal point assumed to be to the right of the value. The largest positive integer with an exponent of 0 that can be represented without loss of precision is 9,223,372,036,854,775,807; the largest negative integer is -9,223,372,036,854,775,808.
- The exponent is stored internally as a signed byte. Its values range from 127 to -128.
This is the base-10 exponent of the value. That is, the value of the number is the significand multiplied by 10 raised to the power of the exponent.

For example, for the ObjectScript literal value 1.23, the significand is 123, and -2 is the exponent.

Thus, the range of numbers that can be represented in Caché native format approximately covers the range 1.0E-128 to 9.22E145. (The first value is the smallest integer with the smallest exponent. The second value is the largest integer with the decimal point moved to the left and the exponent increased correspondingly in the displayed representation.)

All numbers with 18 digits of precision can be represented exactly; numbers which are within the representation bounds of the significand can be accurately represented as 19-digit values.

Note: Caché does not normalize the significand unless necessary to fit the number in decimal format. So numbers with a significand of 123 and an exponent of 1, and a significand of 1230 and an exponent of zero compare as equal.

D.1.2 \$DOUBLE Format

The Caché \$DOUBLE format conforms to [IEEE-754–2019](#), specifically, the 64-bit binary (double-precision) representation. This means it consists of three parts:

- A sign bit
- An 11-bit power of two exponent. The exponent value is biased by 1023, so the internal value of the exponent for the number \$DOUBLE(1.0) is 1023 rather than 0.
- A positive 52-bit fractional significand. Because the significand is always treated as a positive value and normalized, a 1-bit is assumed as the lead binary digit even though it is not present in the significand. Thus, the significand is numerically 53 bits long: the value 1, followed by the implied binary point, followed by the fractional significand. This can be thought of as an integer implicitly divided by 2^{52} .

As an integer, all values between 0 and 9,007,199,254,740,992 can be represented exactly. Larger integers may or may not have exact representations depending on their pattern of bits.

This representation has three optional features that are not available with Caché native format:

- The ability to represent the results of invalid computations (such as taking the square root of a negative number) as a NaN (Not any Number).
- The ability to represent both a +0 and -0.
- The ability to represent infinity.
- The standard provides for representation of numbers smaller than 2^{-1022} . This is done by a technique referred to as a “gradual loss of precision”. Please refer to the [standard](#) for details.

These features are under program control via the **IEEEError()** method of the %SYSTEM.Process class for an individual process or the **IEEEError()** method of the Config.Miscellaneous class for the system as a whole.

Important: Calculations using IEEE binary floating-point representations can give different results for the same IEEE operation. InterSystems has written its own implementations for:

1. Conversions between \$DOUBLE binary floating-point and decimal;
2. Conversion between \$DOUBLE and numeric strings;
3. Comparisons between \$DOUBLE and other numeric types.

This guarantees that when a \$DOUBLE value is inserted into, or fetched from, a Caché data base, the result is the same across all hardware platforms.

However, for all other calculations involving the \$DOUBLE type, Caché uses the vendor-supplied floating-point library subroutines. This means that there can be minor differences between platforms for the same set of operations. In all cases, however, Caché \$DOUBLE calculations equal the local calculations performed on the C double type; that is, the differences between platforms for Caché \$DOUBLE computations are never worse than the differences exhibited by C programs computing IEEE values running on those same platforms.

D.1.3 SQL Representations

The Caché SQL data types DOUBLE and DOUBLE PRECISION represent IEEE floating-point numbers, that is, \$DOUBLE. The SQL FLOAT data type represents standard Caché decimal numbers.

D.2 Choosing a Numeric Format

The choice of which format to use is largely determined by the requirements of the computation. Caché decimal format permits over 18 decimal digits of accuracy while \$DOUBLE guarantees only 15.

In most cases, decimal format is simpler to use and provides more precise results. It is usually preferred for computations involving decimal values (such as currency calculations) because it gives the expected results. Decimal fractions cannot often be represented exactly as binary fractions.

On the other hand, the range of numbers in \$DOUBLE is significantly larger than permitted by native format: 1.0E308 versus 1.0E145. Those applications where the range is a significant factor should use \$DOUBLE.

Applications that will share data externally may also consider maintaining data in \$DOUBLE format because it will not be subject to implicit conversion. Most other systems use the IEEE standard as their representation of binary floating-point numbers because it is supported directly by the underlying hardware architecture. So values in decimal format must be converted before they can be exchanged, for example, via ODBC/JDBC, SQL, or language binding interfaces.

If a \$DOUBLE value is within the bounds defined for Caché decimal numbers, then converting it to decimal and then converting back to a \$DOUBLE value will always yield the same number. The reverse is not true because \$DOUBLE values have less precision than decimal values.

For this reason, InterSystems recommends that computation be done in one representation or the other, when possible. Converting values back and forth between representations may cause loss of accuracy. Most applications can use Caché decimal format for all their computations. The \$DOUBLE format is intended to support those applications that exchange data with systems that use IEEE formats.

The reasons for preferring Caché decimal over \$DOUBLE are:

- Caché decimal has more precision, almost 19 decimal digits compared to less than 16 decimal digits for \$DOUBLE.
- Caché decimal can exactly represent decimal fractions. The value 0.1 is an exact value in Caché decimal; but there is no exact equivalent in binary floating point, so 0.1 must be approximated in \$DOUBLE format.

The advantages of \$DOUBLE over Caché decimal for scientific numbers are:

- \$DOUBLE uses exactly the same representation as the IEEE double precision binary floating point used by most computing hardware.
- \$DOUBLE has a greater range: 1.7E308 maximum for \$DOUBLE and 9.2E145 maximum for Caché decimal.

D.3 Converting Numeric Representations

Beginning in Caché 2007.1, numbers – numeric literals and the results of computations – were automatically converted to a \$DOUBLE representation when the value exceeded the range of a decimal number.

Beginning with Caché 2008.2, only numeric literals are automatically converted to \$DOUBLE. Computational results in decimal that are out of range generate the appropriate error, as discussed later in this appendix.

CAUTION: InterSystems recommends that your application explicitly control conversions between decimal and \$DOUBLE formats.

When converting values from string to number, or when processing written constants when a program is compiled, only the first 38 significant digits can influence the value of the significand. All digits following that will be treated as if they were zero; that is, they will be used in determining the value of the exponent but they will have no additional effect on the significand value.

D.3.1 Strings

D.3.1.1 Strings as Numbers

In Caché, if a string is used in an expression, the value of the string is the value of the longest numeric literal contained in the string starting at the first character. If there is no such literal present, the computed value of the string is zero.

D.3.1.2 Numeric Strings As Subscripts

In computation, there is no difference between the strings “04” and “4”. However, when such strings are used as subscripts for local or global arrays, Caché makes a distinction between them.

In Caché, numeric strings that contain leading zeroes (after the minus sign, if there is one), or trailing zeroes at the end of decimal fractions, will be treated as if they were strings when used as subscripts. As strings, they have a numeric value; they can be used in computations. But as subscripts for local or global variables, they are treated as strings and are collated as strings. Thus, in the list of pairs:

- “4” versus “04”
- “10” versus “10.0”
- “.001” versus “0.001”
- “-.3” versus “-0.3”
- “1” versus “+01”

those on the left are considered numbers when used as subscripts and those on the right are treated as strings. (The form on the left, without the extraneous leading and trailing zero parts, is sometimes referred to as “canonical” form.)

In normal collation, numbers sort before strings as shown in this example,

ObjectScript

```

SET ^| |TEST("2") = "standard"
SET ^| |TEST("01") = "not standard"
SET NF = "Not Found"

WRITE ""2"", ": ", $GET(^| |TEST("2"),NF), !
WRITE 2, ": ", $GET(^| |TEST(2),NF), !
WRITE ""01"", ": ", $GET(^| |TEST("01"),NF), !
WRITE 1, ": ", $GET(^| |TEST(1),NF), !, !
SET SUBS=$ORDER(^| |TEST(""))
WRITE "Subscript Order:", !
WHILE (SUBS '= "") {
    WRITE SUBS, !
    SET SUBS=$ORDER(^| |TEST(SUBS))
}

```

D.3.2 Decimal to \$DOUBLE

Conversion to \$DOUBLE format is done explicitly via the **\$DOUBLE** function. This function also permits the explicit construction of IEEE representations for not-a-number and infinity via the expression, **\$DOUBLE(<S>)** where <S> is:

- the string, “nan” to generate a NaN
- any one of the strings “inf”, “+inf”, “-inf”, “infinity”, “+infinity”, or “-infinity” for infinity.
- the numeric and string literals, -0 and “-0”, respectively

Note: The case of the string, <S>, is ignored on input. On output, only “NAN”, “INF” and “-INF” are produced.

D.3.3 \$DOUBLE to Decimal

Values in \$DOUBLE form are converted to decimal values with the **\$DECIMAL** function. The result of calling the function is a string suitable for conversion to a decimal value.

Note: Although this description assumes the value presented to **\$DECIMAL** is a \$DOUBLE value, this is not a requirement. Any numeric value may be supplied as the argument and the same rules apply for rounding.

D.3.3.1 \$DECIMAL(x)

The single argument form of the function converts the \$DOUBLE value given as its argument to decimal. **\$DECIMAL** rounds the decimal portion of the number to 19 digits. **\$DECIMAL** always rounds to the nearest decimal value.

D.3.3.2 \$DECIMAL(x, n)

The two-argument form allows precise control over the number of digits returned. If *n* is greater than 38, an <ILLEGAL VALUE> error occurs. If *n*, is greater than 0, the value of *x* rounded to *n* significant digits is returned.

When *n* is zero, the following rules are used to determine the value:

- If *x* is an Infinity, return “INF” or “-INF” as appropriate.
- If *x* is a NaN, return “NAN”.
- If *x* is a positive or negative zero, return “0”.
- If *x* can be exactly represented in 20 or fewer significant digits, return the canonical numeric string contains those exact significant digits.
- Otherwise, truncate the decimal representation to 20 significant digits, and
 - If the 20th digit is a “0”, replace it with a “1”;

- b. If the 20th digit is a “5”, replace it with a “6”.

Then, return the resulting string.

This rounding rule involving truncation-to-zero of the 20th digit except when it would inexactly make the 20th digit be a “0” or “5” has these properties:

- If a \$DOUBLE value is different from a decimal value, these two values will always have unequal representation strings.
- When a \$DOUBLE value can be converted to decimal without generating a <MAXNUMBER> error, the result is the same as converting the \$DOUBLE value to a string and then converting that string to a decimal value. There is no possibility of a “double round” error when doing the two conversions.

D.3.4 Decimal to String

Decimal values can be converted to strings by default when they are used as such, for example, as one of the operands to the [concatenation operator](#). When more control over the conversion is needed, use the [\\$FNUMBER](#) function.

D.4 Operations Involving Numbers

D.4.1 Arithmetic

D.4.1.1 Homogeneous Representations

Expressions involving only decimal values will always yield a decimal result. Similarly, expressions with only \$DOUBLE values will always produce a \$DOUBLE result. In addition,

- If the result of a computation involving decimal values overflows, a <MAXNUMBER> error will result. There is no automatic conversion to \$DOUBLE in this case as there is for literals.
- If a decimal expression underflows, 0 is generated as the result of the expression.
- By default the IEEE errors of overflow, divide-by-zero, and invalid-operation will signal the <MAXNUMBER>, <DIVIDE>, and <ILLEGAL VALUE> errors, respectively, rather than generating an Infinity or NaN result. This behavior can be modified by the **IEEEError()** method of the %SYSTEM.Process class for an individual process or the **IEEEError()** method of the Config.Miscellaneous class for the system as a whole.
- The expression 0 ** 0 (decimal) produces the decimal value, 0; but, the expression \$DOUBLE(0) ** \$DOUBLE(0) produces the \$DOUBLE value, 1. The former has always been true in Caché; the latter is required by the IEEE standard.

D.4.1.2 Heterogenous Representations

Expressions involving both decimal and \$DOUBLE representations always produce a \$DOUBLE value. The conversion of the value takes place when it is used. Thus, in the expression

```
1 + 2 * $DOUBLE(4.0)
```

Caché first adds 1 and 2 together as decimal values. Then it converts the result, 3, to \$DOUBLE format and does the multiplication. The result is \$DOUBLE(12).

D.4.1.3 Rounding

When necessary, numeric results are rounded to the nearest representable value. When the value to be rounded is equally close to two available values, then:

- \$DOUBLE values are rounded to even as defined in the IEEE standard
- Decimal values are rounded away from zero, that is toward a larger value (in absolute terms)

D.4.2 Comparison

D.4.2.1 Homogeneous Representations

Comparisons between \$DOUBLE(+0) and \$DOUBLE(-0) treat these values as equal. This follows the IEEE standard. This is the same as in Caché decimal because, when either \$DOUBLE(+0) or \$DOUBLE(-0) is converted to a string, the result in both cases is “0”.

Comparisons between \$DOUBLE(“nan”) and any other numeric value — including \$DOUBLE(“nan”) — will say these values are not greater than, not equal, and not less than. This follows the IEEE standard. This is a departure from usual Caché rule that says the equality comparison is done by converting to strings and checking the strings for equality.

Note: The expression, “nan”, is equal to \$DOUBLE(“nan”) because the comparison is done as a string compare.

Note: However, \$LISTSAME considers a list component containing \$DOUBLE(“nan”) to be the same as a list component containing \$DOUBLE(“nan”). This is the only place where \$LISTSAME considers what should be unequal values to be equal.

D.4.2.2 Heterogeneous Representations

Comparisons between a decimal value and \$DOUBLE value are fully accurate. The comparisons are done without any rounding of either value. If only finite values are involved then these comparisons get the same answer that would result if both values were converted to strings and those strings were compared using the default collation rules.

Comparison involving the operators <, <=, >, and >= always produce a boolean result, 0 or 1, as a decimal value. If one of the operands is a string, that operand is converted to a decimal value before the comparison is performed. Other numeric operands are not converted. As noted, the comparison of mixed numeric types is done with full accuracy and no conversion.

In the case of the string comparison operators (=, !=,],], [, [,],], and so on), any numeric operand is first converted to a string before the comparison is done.

D.4.2.3 Less-Than Or Equal, Greater-Than Or Equal

In Caché, the operators “<=” and “>=” are treated as synonyms for the operators “>” and “<”, respectively.

CAUTION: If the operators “<=” or “>=” are used in comparisons where either or both of the operands may be NaNs, the results will be different from those mandated by the IEEE standard.

The expression “A >= B” when either A and/or B is a NaN is interpreted as follows:

1. The expression is transformed to “A > B”.
2. It is further transformed to “!(A > B)”.
3. As noted previously, comparisons involving NaNs give results that are (a) not equal, (b) not greater-than, and (c) not less-than, so the expression in parenthesis results in a value of false.
4. The negation of that value results in a value of true.

Note: The expression “A >= B” can be rewritten to provide the IEEE expected results if it is expressed as “((A > B) | (A = B))”.

D.4.3 Boolean Operations

For boolean operations and, or not, nor, nand and so on) any string operand is converted to decimal. Any numeric operand (decimal or \$DOUBLE) is left unchanged.

A numeric value that is zero is treated as FALSE; all other numeric values (including \$DOUBLE(“nan”) and \$DOUBLE(“inf”)) are treated as TRUE. The result is 0 or 1 (as decimal.)

Note: \$DOUBLE(-0) is also false.

D.5 Summary of Changes Introduced in Version 2008.2

The following are the significant changes to numeric processing for Caché 2008.2 and following:

1. Conversion of \$DOUBLE values to a decimal string representation will now provide 20 significant digits. In prior releases, the conversion yielded 15.
2. Computational errors involving \$DOUBLE values result in Caché errors (<MAXNUMBER>, <DIVIDE>, <VALUE>) by default. They do not result in the IEEE infinity or NaN values. This can be changed via the **IEEEError()** method of the %SYSTEM.Process class for an individual process or the **IEEEError()** method of the Config.Miscellaneous class for the system as a whole.
3. There is an automatic conversion of numeric literal values outside the range of Caché decimal to \$DOUBLE. This is true only for literals. It does *not* happen for the results of computations.

D.6 See Also

For more information, see the following sources:

- The [IEEE-754–2019](#) standard.
- “[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#),” by David Goldberg, published in the March, 1991 issue of *Computing Surveys*.