
# Using Caché Objects

Version 2018.1
2024-05-02

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

*Using Caché Objects*
Caché Version 2018.1 2024-05-02


For Support questions about any InterSystems products, contact:

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

# Table of Contents

# List of Figures

# List of Tables

# About This Book

This book is a guide to creating and using classes in Caché, particularly object classes and persistent classes. It consists of the following chapters:

- Introduction to Caché Objects
- Defining and Compiling Classes
- Package Options
- Defining and Referring to Class Parameters
- Defining and Calling Methods
- Working with Registered Objects
- Introduction to Persistent Objects
- Working with Persistent Objects
- Defining Persistent Classes
- Defining and Using Literal Properties
- Working with Collections
- Working with Streams
- Defining and Using Object-Valued Properties
- Defining and Using Relationships
- Other Options for Persistent Classes
- Defining Method and Trigger Generators
- Defining and Using Class Queries
- Defining and Using XData Blocks
- Defining Class Projections
- Defining Data Type Classes
- Implementing Callback Methods
- Using and Overriding Property Methods
- Implementing Dynamic Dispatch

This book also includes the following appendices:

- Object-Specific ObjectScript Features
- Using the Caché Populate Utility
- Using the %Dictionary Classes
- Using the Object Synchronization Feature

For a detailed outline, see the table of contents.

For information about related topics, see the following documents:

- *Caché Programming Orientation Guide* is an orientation guide for programmers who are new to Caché or who are familiar with only some kinds of Caché programming.
- *The Caché Class Definition Reference* provides detailed reference information about how to define classes and their members.
- *Using Caché ObjectScript* describes concepts and how to use the ObjectScript language.
- *Using Caché Globals* describes the underlying data storage mechanisms that Caché uses.

For general information, see *Using InterSystems Documentation*.

# 1

# Introduction to Caché Objects

This manual describes the various aspects of Caché objects. You may also find the *Caché Web Applications Tutorial* to be a useful introduction to the topic.

Caché object technologies give application developers the means to easily create high performance, object-based, database applications.

The features of Caché objects include:

- A powerful object model that includes inheritance, properties, methods, collections, relationships, user-defined data types, and streams.
- A flexible object persistence mechanism that allows objects to be stored within the native Caché database as well as external relational databases.
- Control over the database aspects of persistent classes including indices, constraints, and referential integrity.
- An easy-to-use transaction and concurrency model that includes the ability to load objects by navigation—simply referring to an object can "swizzle" it into memory from the database.
- Automatic integration with Caché SQL via the Caché unified data architecture.
- Interoperability with Java, C++, and ActiveX.
- Automatic XML support.
- A powerful, multi-user object development environment: Studio.

You can use Caché objects in many ways including:

- To define the database and/or business components of a transaction processing application.
- To create a web-based user interface using Caché Server Pages.
- To define object-based stored procedures that are callable from ODBC or JDBC.
- To provide object/relational access to legacy applications.

## 1.1 Caché Objects Architecture

Caché object technology contains the following major components:

- The *class dictionary* — A repository of class definitions (often known as metadata), each of which describes a specific class. This repository is stored within a Caché database. The class dictionary is also used by the Caché SQL engine and is responsible for maintaining synchronized object and relational access to Caché data.
- The *class compiler* — A set of programs that convert class definitions into executable code.
- The *object runtime system* — A set of features built into the Caché virtual machine that support object operations (such as object instantiation, method invocation, and polymorphism) within a running program.
- The *Caché class library* — A set of prebuilt classes that come with every Caché installation. This includes classes that are used to provide behaviors for user-defined classes (such as persistence or data types) as well as classes that are intended for direct use within applications (such as email classes).
- The various *language bindings* — A combination of code generators and runtime components that provide external access to Caché objects. These bindings include the Caché Java binding, the Caché ActiveX binding, and the Caché C++ binding.
- The various *gateways* — Server-side components that give Caché objects access to external systems. These gateways include the Caché SQL Gateway and the Caché Activate ActiveX Gateway.

# 1.2 Class Definitions and the Class Dictionary

Every class has a *definition* that specifies what members (properties, methods, and so on) it contains as well as class-wide characteristics (such as superclasses). These definitions are contained within the class dictionary, which is itself stored within the Caché database.

## 1.2.1 Creating Class Definitions

You can create class definitions in many ways:

- Using Studio. The primary means of working with Caché class definitions is with the Studio Development Environment.
- Using XML. Class definitions have an external, XML-based representation. Typically this format is used for storing class definitions externally (such as in source control systems), deploying applications, or simply for sharing code. You can also create new class definitions programmatically by simply generating the appropriate XML class definition file and loading it into a Caché system.
- Using an API. Caché includes a set of class definition classes that provide object access to the class dictionary. You can use these to observe, modify, and create class definitions.
- Using SQL DDL. Any relational tables defined by DDL statements are automatically converted to equivalent class definitions and placed within the class dictionary.

## 1.2.2 The Class Dictionary

Every Caché namespace contains its own class dictionary which defines the available classes for that namespace. There is a special "CACHELIB" database, installed as part of Caché, that contains the definitions and executable code for the classes of the Caché class library. These classes are referred to as *system* classes and all are part of packages whose names start with a "%" character, such as %Library.Persistent (the names of members of the %Library package can be abbreviated, so that %String is an abbreviation for %Library.String).

Every Caché namespace is automatically configured so that its class dictionary, in addition to containing its own classes, has access to the system class definitions and code within the CACHELIB database. By this mechanism, all namespaces can make direct use of the classes in the Caché class library.

The class dictionary contains two distinct types of data:

- Definition data — The actual class definitions that users create.
- Compilation data — Data generated as a result of compiling class definitions is also stored. This data includes the results of inheritance resolution; that is, it lists all the defined and inherited members for a given class. The class compiler uses this to make other compilation more efficient; applications can also use it (via the appropriate interface) to get runtime information about class members.

The class dictionary stores its data in a set of globals (persistent arrays) whose names start with ^*odd*. The structure of these arrays may change with new versions of Caché, so applications should never directly observe or modify these structures.

# 1.3 The Caché Class Library

The Caché class library contains a set of prebuilt classes. It is automatically installed with every Caché system within the CACHELIB database. You can view the contents of the Caché class library using the online class documentation system provided with Caché.

The Caché class library contains a number of packages, each of which contains a family of classes. Some of these are internal and Caché objects uses them as part of its implementation. Other classes provide features that are designed for use in applications.

The main packages within the Caché class library include:

***Table 1–1: Caché Class Library Packages***

| Package | Description |
|---|---|
| %Activate | Classes used by the Caché Activate ActiveX gateway. See *Using the Caché ActiveX Gateway*. |
| %Compiler | Internal classes used by the class compiler. |
| %CSP | Classes used by Caché Server Pages. See *Using Caché Server Pages (CSP)*. |
| %csr | A set of generated internal classes that implement the standard CSP rules. |
| %Library | The core set of Caché "behavioral" classes (such as %Persistent). Also includes the various data types, collections, and stream classes. |
| %Net | A set of classes providing various Internet-related features (such as email, HTTP requests, and so on). See *Using Caché Internet Utilities*. |
| %Projection | A set of projection classes that generate client-side code for user classes. See the chapter "Class Projections." |
| %SOAP | Classes you can use to create web services and web clients within Caché. See *Creating Web Services and Web Clients in Caché*. |
| %SQL | Internal classes used by Caché SQL. |
| %Studio | Internal classes used by Studio. |
| %SYSTEM | The various system API classes accessible via the *$System* special variable. |
| %XML | Classes used to provide XML and SAX support within Caché. For information, see *Projecting Objects to XML* and *Using Caché XML Tools*. |

For a comprehensive list of options, see the table of contents of the *InterSystems Programming Tools Index*.

# 1.4 Development Tools

Caché includes a number of tools for developing object-based applications. In addition, it is quite easy to use Caché with other development environments.

## 1.4.1 Studio

Studio is an integrated, visual development environment for creating Caché class definitions. See *Using Studio* for more details.

## 1.4.2 SQL-Based Development

It is possible to develop Caché applications using SQL-based tools, since Caché automatically creates a class definition from any relational tables defined using SQL DDL statements. This approach does not exploit the full power of objects because you are limited by what DDL can express; however, it is useful for migrating legacy applications.

## 1.4.3 XML-Based Development

It is possible to develop class definitions as XML documents and load them into Caché. You can do this using either the Caché-specific XML format for class definitions or you can use an XML schema representation.

# 1.5 User Interface Development and Client Connectivity

Caché object technology supports connections with the Caché user-interface development tools as well as its connectivity tools for interoperating with other systems. For a comprehensive list of options, see the table of contents of the *InterSystems Programming Tools Index*.

# 2

# Defining and Compiling Classes

This chapter describes the basics of defining and compiling classes. It discusses the following topics:

- Introduction to terminology
- Kinds of classes
- Kinds of class members
- Basic information on defining a class
- Naming conventions
- Inheritance
- Compiler keywords
- How to create class documentation
- How to compile classes
- How to make classes deployed

When viewing this book online, use the preface of this book to quickly find other topics.

## 2.1 Introduction to Terminology

The following shows a simple Caché class definition, with some typical elements:

### Class Definition

```
Class Demo.MyClass Extends %RegisteredObject
{

Property Property1 As %String;

Property Property2 As %Numeric;

Method MyMethod() As %String
{
   set returnvalue=..Property1_..Property2
   quit returnvalue
}

}
```

Note the following points:

- The full class name is `Demo.MyClass`, the package name is `Demo`, and the short class name is `MyClass`.
- This class extends the class %RegisteredObject. Equivalently, this class inherits from %RegisteredObject.

  %RegisteredObject is the superclass of this class, or this class is a subclass of %RegisteredObject. A Caché class can have multiple superclasses, as this chapter later discusses.

  The superclass(es) of a class determine how the class can be used.
- This class defines two properties: `Property1` and `Property2`. Property `Property1` is of type %String, and Property `Property2` is of type %Numeric.
- This class defines one method: `MyMethod()`, which returns a value of type %String.

This class refers to several system classes provided by Caché. These classes are %RegisteredObject (whose full name is %Library.RegisteredObject), %String (%Library.String), and %Numeric (%Library.Numeric). %RegisteredObject is a key class in Caché, because it defines the object interface. It provides the methods you use to create and work with object instances. %String and %Numeric are data type classes. As a consequence, the corresponding properties hold literal values (rather than other kinds of values).

# 2.2 Kinds of Classes

Caché provides a large set of class definitions that your classes can use in the following general ways:

- You can use Caché classes as superclasses for your classes.
- You can use Caché classes as values of properties, values of arguments to methods, values returned by methods, and so on.
- Some Caché classes simply provide specific APIs. You typically do not use these classes in either of the preceding ways. Instead you write code that calls methods of the API.

The most common choices for superclasses are as follows:

- %RegisteredObject — This class represents the object interface in its most generic form.
- %Persistent — This class represents a persistent object. In addition to providing the object interface, this class provides methods for saving objects to the database and reading objects from the database.
- %SerialObject — This class represents an object that can be embedded in (serialized within) another object.
- Subclasses of any of the preceding classes.
- None — It is not necessary to specify a superclass when you create a class.

The most common choices for values of properties, values of arguments to methods, values returned by methods, and so on are as follows:

- Object classes (the classes contained in the previous list)
- Data type classes
- Collection classes
- Stream classes

Later chapters of this book discuss these categories of classes.

## 2.2.1 Object Classes

The phrase *object class* refers to any subclass of %RegisteredObject. With an object class, you can create an instance of the class, specify properties of the instance, and invoke methods of the instance. A later chapter describes these tasks (and provides information that applies to all object classes).

The generic term *object* refers to an instance of an object class.

There are three general categories of object classes:

- *Transient object classes* or *registered object classes* are subclasses of %RegisteredObject but not of %Persistent or %SerialObject (see the following bullets).

  For details, see "Working with Registered Objects."

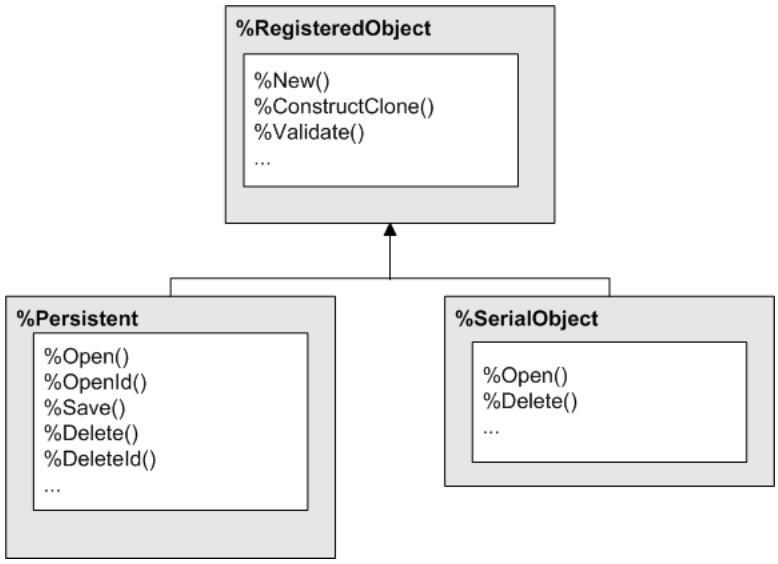
- *Persistent classes* are subclasses of %Persistent, which is a direct subclass of %RegisteredObject. The %Persistent class includes the behavior of %RegisteredObject and adds the ability to save objects to disk, reopen them, and so on.

  For details, see the chapter "Introduction to Persistent Objects" and the chapters that follow it.

- *Serial classes* are subclasses of %SerialObject, which is a direct subclass of %RegisteredObject. The %SerialObject class includes the behavior of %RegisteredObject and adds the ability to create a string that represents the state of the object, for inclusion as a property within another object (usually either a transient object or a persistent object). The phrase *serializing an object* refers to the creation of this string.

  For details, see the chapter "Defining and Using Object-Valued Properties."

The following figure shows the inheritance relationship among these three classes. The boxes list some of the methods defined in the classes:




Collection classes and stream classes are object classes with specialized behavior.

## 2.2.2 Data Type Classes

The phrase *data type class* refers to any class whose ClassType keyword equals `datatype` or any subclass of such a class. These classes are not object classes (a data type class cannot define properties, and you cannot create an instance of the class). The purpose of a data type class (more accurately a data type generator class) is to be used as the type of a property of an object class.

# 2.3 Kinds of Class Members

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, in most cases.
- Methods — Caché supports two types of methods: instance methods and class methods. An *instance method* is invoked from a specific instance of a class and performs some action related to that instance; this type of method is useful only in object classes. A *class method* is a method that can be invoked whether or not an instance of its class is in memory; this type of method is called a *static method* in other languages.
- Properties — A property contains data for an instance of the class. Properties are useful only in object classes. The following subsection provides more information.
- 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. Often (but not necessarily), you define class queries in a persistent class, to perform queries on the stored data for that class. You can, however, define class queries in any class.
- Other kinds of class members that are relevant only for persistent classes:
  - Storage definitions
  - Indices
  - Foreign keys
  - SQL triggers
- XData blocks — An XData block is a named unit of data defined within the class, typically for use by a method in the class. These have many possible applications.
- Projections — A class projection provides a way to extend the behavior of the class compiler.

  The projection mechanism is used by the Java and C++ projections; hence the origin of the term *projection*.

## 2.3.1 Kinds of Properties

Formally, there are two kinds of properties: attributes and relationships.

Attributes hold values. Attribute properties are usually referred to simply as *properties*. Depending on the property definition, the value that it holds can be any of the following:

- A literal value such as `"MyString"` and 1. Properties that hold literal values are based on data type classes and are also called data type properties. See the chapter "Defining and Using Literal Properties."
- A stream. A stream is a Caché object that contains a value that would be too long for a string. See the chapter "Working with Streams."
- A collection. Caché provides the ability to define a property as either a list or an array. The list or array items can be literal values or can be objects. See the chapter "Working with Collections."
- Some other kind of object. See the chapter "Defining and Using Object-Valued Properties."

Relationships hold associations between objects. Relationship properties are referred to as *relationships*. Relationships are supported only in persistent classes. See the chapter "Defining and Using Relationships."

# 2.4 Defining a Class: The Basics

This section discusses basic class definitions in more detail. It discusses the following topics:

- Choosing a superclass
- Specifying class keywords
- Include files
- Introduction to defining class parameters
- Introduction to defining properties
- Introduction to defining methods

Typically, you use Studio to define classes. You can also define classes programmatically using the Caché class definition classes or via an XML class definition file. If you define an SQL table using SQL DDL statements, the system creates a corresponding class definition.

## 2.4.1 Choosing a Superclass

When you define a class, one of your earliest design decisions is choosing the class (or classes) which to base your class. If there is only a single superclass, include `Extends` followed by the superclass name, at the start of the class definition.

### Class Definition

```
Class Demo.MyClass Extends Superclass
{

//...

}
```

If there are multiple superclasses, specify them as a comma-separated list, enclosed in parentheses.

### Class Definition

```
Class Demo.MyClass Extends (Superclass1, Superclass2, Superclass3)
{

//...

}
```

It is not necessary to specify a superclass when you create a class. It is common to use %RegisteredObject as the superclass even if the class does not represent any kind of object, because doing so gives your class access to many commonly used macros, but you can instead directly include the include files that contain them.

## 2.4.2 Include Files

When you create a class that does not extend %RegisteredObject or any of its subclasses, you might want to include the following include files:

- %occStatus.inc, which defines macros to work with %Status values.
- %occMessages.inc, which defines macros to work with messages.

  For details on the macros defined by these include files, see "Using System-supplied Macros" in *Using Caché ObjectScript*.

If your class *does* extend %RegisteredObject or any of its subclasses, these macros are available automatically.

You can also create your own include files and include them in class definitions as needed.

To include an include file at the beginning of a class definition, use syntax of the following form. Note that you must omit the `.inc` extension of the include file:

```
Include MyMacros
```

For example:

```
Include %occInclude

Class Classname
{
}
```

To include multiple include files at the beginning of a class definition, use syntax of the following form:

```
Include (MyMacros, YourMacros)
```

Note that this syntax does not have a leading pound sign (in contrast to the syntax required in a routine). Also, the `Include` directive is not case-sensitive, so you could use `INCLUDE` instead, for example. The include file name is case-sensitive.

See also the reference section on #include in *Using Caché ObjectScript*.

## 2.4.3 Specifying Class Keywords

In some cases, it is necessary to control details of the code generated by the class compiler. For one example, for a persistent class, you can specify an SQL table name, if you do not want to (or cannot) use the default table name. For another example, you can mark a class as final, so that subclasses of it cannot be created. The class definitions support a specific set of keywords for such purposes. If you need to specify class keywords, include them within square brackets after the superclass, as follows:

```
Class Demo.MyClass Extends Demo.MySuperclass [ Keyword1, Keyword2, ...]
{

//...

}
```

For example, the available class keywords include Abstract and Final. For an introduction, see "Compiler Keywords," later in this chapter. Caché also provides specific keywords for each kind of class member.

## 2.4.4 Introduction to Defining Class Parameters

A class parameter defines a constant value for all objects of a given class. To add a class parameter to a class definition, add an element like one of the following to the class:

### Class Member

```
Parameter PARAMNAME as Type;

Parameter PARAMNAME as Type = value;

Parameter PARAMNAME as Type [ Keywords ] = value;
```

*Keywords* represents any parameter keywords. For an introduction to keywords, see "Compiler Keywords," later in this chapter. For parameter keywords; see "Parameter Keywords" in the *Caché Class Definition Reference*. These are optional.

## 2.4.5 Introduction to Defining Properties

An object class can include properties.

To add a property to a class definition, add an element like one of the following to the class:

### Class Member

```
Property PropName as Classname;

Property PropName as Classname [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) ;
```

*PropName* is the name of the property, and *Classname* is an optional class name (if you omit this, the property is assumed to be of type %String).

*Keywords* represents any property keywords. For an introduction to keywords, see "Compiler Keywords," later in this chapter. For property keywords; see "Property Keywords" in the *Caché Class Definition Reference*. These are optional.

Depending on the class used by the property, you might also be able to specify *property parameters*, as shown in the third and fourth variations.

Notice that the property parameters, if included, are enclosed in parentheses and precede any property keywords. Also notice that the property keywords, if included, are enclosed in square brackets.

## 2.4.6 Introduction to Defining Methods

You can define two kinds of methods in Caché classes: class methods and instance methods.

To add a class method to a class definition, add an element like the following to the class:

```
ClassMethod MethodName(arguments) as Classname [ Keywords]
{
//method implementation
}
```

*MethodName* is the name of the method and *arguments* is a comma-separated list of arguments. *Classname* is an optional class name that represents the type of value (if any) returned by this method. Omit the `As` *Classname* part if the method does not return a value.

*Keywords* represents any method keywords. For an introduction to keywords, see "Compiler Keywords," later in this chapter. For method keywords, see "Method Keywords" in the *Caché Class Definition Reference*. These are optional.

To add an instance method, use the same syntax with `Method` instead of `ClassMethod`:

```
Method MethodName(arguments) as Classname [ Keywords]
{
//method implementation
}
```

Instance methods are relevant only in object classes.

# 2.5 Naming Conventions

Class and class members follow specific naming conventions. These are detailed in this section.

## 2.5.1 Rules for Class and Class Member Names

This section describes the rules for class and member names, such as maximum length, allowed characters, and so on. A full class name includes its package name, as described in the next section.

Every identifier must be unique within its context (that is, no two classes can have the same name). Caché has the following limits on package, class, and member names:

- Each package name can have up to 189 unique characters.
- Each class name can have up to 60 unique characters.
- Each method and property name can have up to 180 unique characters. See the section "Class Member Names" for more details.
- 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, the class name, the package name, and any separators) must be 220 characters or fewer.
- Each name can include Unicode characters.

Identifiers preserve case: you must exactly match the case of a name; at the same time, two classes cannot have names that differ only in case. For example, the identifiers "id1" and "ID1" are considered identical for purposes of uniqueness.

Identifiers must start with an alphabetic character, though they may contain numeric characters after the first position. Identifiers cannot contain spaces or punctuation characters with the exception of package names which may contain the "." character. On a Unicode system, identifiers may contain Unicode characters.

Certain identifiers start with the "%" character; this identifies a system item. For example, many of the methods and packages provided with the Caché library start with the "%" character.

Member names can be delimited, which allows them to include characters that are otherwise not permitted. To create a delimited member name, use double quotes for the first and last characters of the name. For example:

#### Class Member

```
Property "My Property" As %String;
```

For more details on system identifiers, see the appendix "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*.

## 2.5.2 Class Names

Every class has a name that uniquely identifies it. A full class name consists of two parts: a package name and a class name: the class name follows the final "." character in the name. A class name must be unique within its package; a package name must be unique within a Caché namespace. For details on packages, see the chapter "Packages."

Because persistent classes are automatically projected as SQL tables, a class definition must specify a table name that is *not* an SQL reserved word; if the name of a persistent class is an SQL reserved word, then the class definition must also specify a valid, non-reserved word value for its SQLTableName keyword.

## 2.5.3 Class Member Names

Every class member (such as a property or method) must have a name that is unique within its class and with a maximum length of 180 characters. Further, a member of a persistent cannot use an SQL reserved word as its identifier. It can define an alias, however, using the SQLName or SQLFieldName keyword of that member (as appropriate).

**Important:** InterSystems strongly recommends that you do not give two members the same name. This can have unexpected results.

# 2.6 Inheritance

A Caché class can inherit from already existing classes. If one class inherits from another, the inheriting class is known as a *subclass* and the class or classes it is derived from are known as *superclasses*.

The following shows an example class definition that uses two superclasses:

### Class Definition

```
Class User.MySubclass Extends (%Library.Persistent, %Library.Populate)
{
}
```

**Note:** The syntax shown here corresponds to the Super keyword, which is visible in the Studio Inspector and in class definitions exported as XML.

In addition to a class inheriting methods from its superclasses, the properties inherit additional methods from system property behavior classes and, in the case of a data type attribute, from the data type class.

For example, if there is a class defined called Person:

### Class Definition

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

It is simple to derive a new class, Employee, from it:

### Class Definition

```
Class MyApp.Employee Extends Person
{
Property Salary As %Integer;
Property Department As %String;
}
```

This definition establishes the Employee class as a subclass of the Person class. In addition to its own class parameters, properties, and methods, the Employee class includes all of these elements from the Person class.

## 2.6.1 Use of Subclasses

You can use a subclass in any place in which you might use its superclass. For example, using the above defined Employee and Person classes, it is possible to open an Employee object and refer to it as a Person:

### ObjectScript

```
 Set x = ##class(MyApp.Person).%OpenId(id)
 Write x.Name
```

We can also access Employee-specific attributes or methods:

### ObjectScript

```
Write x.Salary // displays the Salary property (only available in Employee instances)
```

## 2.6.2 Primary Superclass

The leftmost superclass that a subclass extends is known as its *primary superclass*. A class inherits all the members of its primary superclass, including applicable class keywords, properties, methods, queries, indices, class parameters, and the parameters and keywords of the inherited properties and inherited methods. Except for items marked as *Final*, the subclass can override (but not delete) the characteristics of its inherited members.

See the next section for more details about multiple inheritance.

## 2.6.3 Multiple Inheritance

By means of multiple inheritance, a class can inherit its behavior and class type from more than one superclass. To establish multiple inheritance, list multiple superclasses within parentheses. The leftmost superclass is the primary superclass.

For example, if class X inherits from classes A, B, and C, its definition includes:

### Class Definition

```
Class X Extends (A, B, C)
{
}
```

The default inheritance order for the class compiler is from left to right, which means that differences in member definitions among superclasses are resolved in favor of the leftmost superclass (in this case, A superseding B and C, and B superseding C.)

Specifically, for class X, the values of the class parameter values, properties, and methods are inherited from class A (the first superclass listed), then from class B, and, finally, from class C. X also inherits any class members from B that A has not defined, and any class members from C that neither A nor B has defined. If class B has a class member with the same name as a member already inherited from A, then X uses the value from A; similarly, if C has a member with the same name as one inherited from either A or B, the order of precedence is A, then B, then C.

Because left-to-right inheritance is the default, there is no need to specify this; hence, the previous example class definition is equivalent to the following:

### Class Definition

```
Class X Extends (A, B, C) [ Inheritance = left ]
{
}
```

To specify right-to-left inheritance among superclasses, use the Inheritance keyword with a value of `right`:

### Class Definition

```
Class X Extends (A, B, C) [ Inheritance = right ]
{
}
```

With right-to-left inheritance, if multiple superclasses have members with the same name, the superclass to the right takes precedence.

**Note:** Even with right-to-left inheritance, the leftmost superclass (sometimes known as the first superclass) is still the primary superclass. This means that the subclass inherits only the class keyword values of its leftmost superclass — there is no override for these.

For example, in the case of class X inheriting from classes A, B, and C with right-to-left inheritance, if there is a conflict between a member inherited from class A and one from class B, the member from class B overrides (replaces) the previously inherited member; likewise for the members of class C in relation to those of classes A and B. The class keywords for class X come exclusively from class A. (This is why extending classes A and B — in that order — with left-to-right inheritance is not the same as extending classes B and A — in that order — with right-to-left inheritance; the keywords are inherited from the leftmost superclass in either definition, which makes the two cases different.)

**Important:** Before version 2010.1 of Caché, inheritance order was always right-to-left and could not be changed. Classes from an older instance that has upgraded will automatically continue to use right-to-left inheritance due to a class dictionary upgrade. Hence, existing code does not require any changes, even though new classes use left-to-right inheritance by default from 2010.1 onward.

### 2.6.4 Additional Topics

Also see "%ClassName() and the Most Specific Type Class (MSTC)" in the chapter "Working with Registered Objects."

## 2.7 Introduction to Compiler Keywords

As shown in "Defining a Class: The Basics," you can include keywords in a class definition or in the definition of a class member. These keywords, also known as *class attributes*, generally affect the compiler. This section introduces some common keywords and discusses how Caché presents them.

### 2.7.1 Example

The following example shows a class definition with some commonly used keywords:

```
/// This sample persistent class represents a person.
Class MyApp.Person Extends %Persistent [ SqlTableName = MyAppPerson ]
{

/// Define a unique index for the SSN property.
Index SSNKey On SSN [ Unique ];

/// Name of the person.
Property Name As %String [ Required ];

/// Person's Social Security number.
Property SSN As %String(PATTERN = "3N1""-""2N1""-""4N") [ Required ];

}
```

This example shows the following keywords:

- For the class definition, the Extends keyword specifies the superclass (or superclasses) from which this class inherits.

  Note that the Extends keyword has a different name when you view the class in other ways; see the next section.
- For the class definition, the SqlTableName keyword determines the name of the associated table, if the default name is not to be used. This keyword is meaningful only for persistent classes, which are described later in this book.
- For the index definition, the Unique keyword causes Caché to enforce uniqueness on the property on which the index is based (SSN in this example).
- For the two properties, the Required keyword causes Caché to require non-null values for the properties.

*PATTERN* is not a keyword but instead is a property parameter; notice that *PATTERN* is enclosed in parentheses, rather than square brackets.

Later chapters of this book discuss many additional keywords, but not all of them. Apart from keywords related to storage (which are not generally documented), you can find details on the keywords in the *Caché Class Definition Reference*. The reference information demonstrates the syntax that applies when you view a class in the usual edit mode.

## 2.7.2 Presentation of Keywords and Their Values

In many but not all cases, when you specify a keyword for a class definition or for a class member, you add an element of one of the following forms to the class or class member:

- `[ keyword ]`

  Specifies the keyword as true.

- `[ Not keyword ]`

  Specifies the keyword as false.

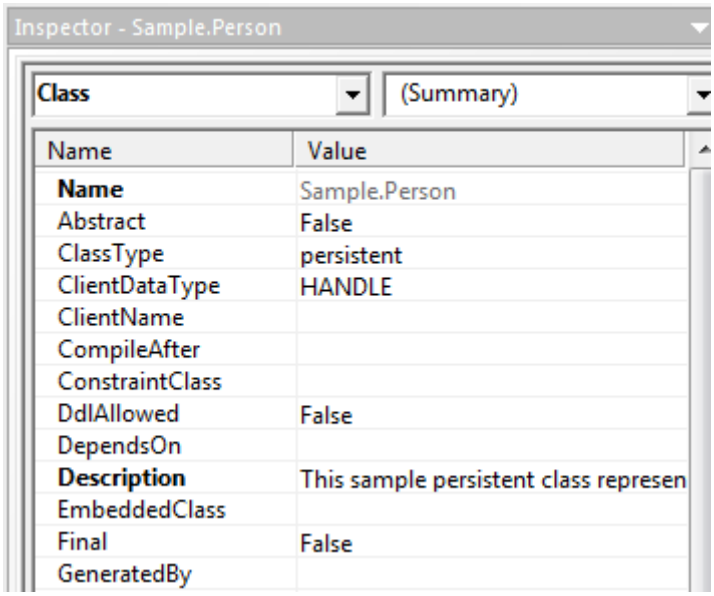
- `[ keyword=value ]`

  Specifies the keyword as the given value.

In the Studio Inspector, the compiler keywords and their values are presented differently. For example, consider the following class definition:

```
/// This sample persistent class represents a person.
/// <p>Maintenance note: This class is used by some of the bindings samples.
Class Sample.Person Extends (%Persistent, %Populate, %XML.Adaptor)
{

...
```

For this class, the Studio Inspector displays the following table of keywords:




Notice that both Name and Description are keywords. If you edit Description in the Inspector, Studio updates the comments in the class definition, and vice versa. Similarly, there is a keyword named Super, which specifies the superclasses of this class. If you edit that, Studio updates the `Extends` part of the class definition.

The Studio Inspector has similar behavior when you display a class member. In that case, the Inspector window displays a table of all the member keywords and the values of those keywords for the currently selected member. (For a property, the Inspector window also lists the available property parameters and their current values.)

When you export a class definition to XML, the exported file looks like the following:

```
<Export generator="Cache" version="25" zv="Cache for Windows (x86-64) 2015.1 (Build 416U)" ts="2014-12-19
 15:27:27">
<Class name="Sample.Person">
<Description><![CDATA[
This sample persistent class represents a person.
<p>Maintenance note: This class is used by some of the bindings samples.]]></Description>
<Super>%Persistent,%Populate,%XML.Adaptor</Super>
<TimeChanged>63540,49568.139638</TimeChanged>
<TimeCreated>59269,38836.623</TimeCreated>

<Parameter name="EXTENTQUERYSPEC">
<Default>Name,SSN,Home.City,Home.State</Default>
</Parameter>

...
```

Most of the XML elements in this file correspond to the compiler keywords.

When you access a class definition programmatically, the class definition instance contains properties that correspond to the keywords. For information on accessing class definitions programmatically, see the chapter "Using the %Dictionary Classes."

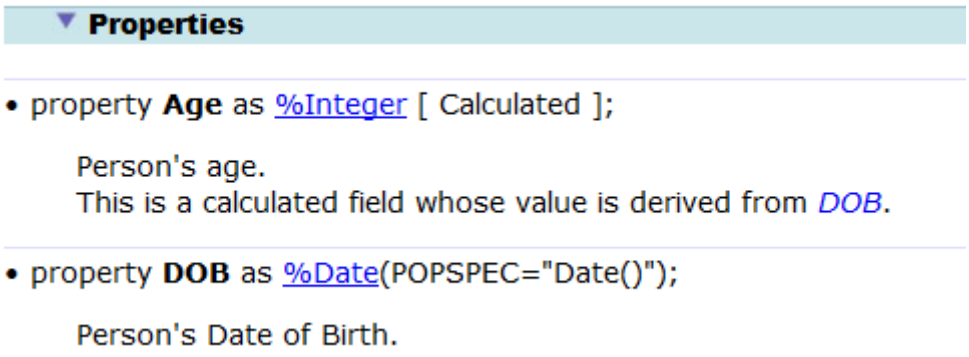
# 2.8 Creating Class Documentation

Caché provides a web page called the *InterSystems Class Reference*, which displays automatically generated reference information for the classes provided by InterSystems, as well as for classes you create. Informally, the Class Reference is known as *Documatic*, because it is generated by the class %CSP.Documatic.

This section introduces the Class Reference and explains how to create your own documentation and how to include HTML markup.

## 2.8.1 Introduction to the Class Reference

The purpose of the Class Reference is to advertise, to other programmers, which parts of a class can be used, and how to use them. The following shows an example:




This reference information shows the definitions of class members, but not their actual implementations. For example, it shows 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.

## 2.8.2 Creating Documentation to Include in the Class Reference

To create documentation to include in the Class Reference, create comments within the class definitions — specifically comments that start with `///`. If you precede the class declaration with such comments, the comments are shown at the top of the page for the class. If you precede a given class member with such comments, the comments are shown after the generated information for that class member. Once you compile the class, you can view its generated class documentation the next time you open the *Class Reference* documentation. If you add no Class Reference comments, items that you add to a class or package appear appropriately in the lists of class or package contents, but without any explanatory text.

You can extend any existing Class Reference comments from within Studio, either by editing the **Description** field for a class in the Studio Inspector window, or by adding specially formatted lines to the class code. The syntax rules for Class Reference comments are strict:

- The length of the Class Reference comment (all lines combined) must be less than the maximum string length for your system; see "Long String Limit" in the *Caché Programming Orientation Guide*.
- All Class Reference comments that describe a class or class member must appear in a consecutive block immediately before the declaration of the item that they describe.
- Each line in the block of comments must start with three slashes: `///`

  **Tip:** Note that, by default, the presentation combines the text of all the `///` lines and treats the result as single paragraph. You can insert HTML line breaks (<br>). Or you can use HTML formatting (such as  and ), as discussed in the subsection.

- The three slashes must begin at the first (left-most) position in the line.
- No blank lines are allowed within Class Reference comments.
- No blank lines are allowed between the last line of the Class Reference comments and the declaration for the item that they describe.

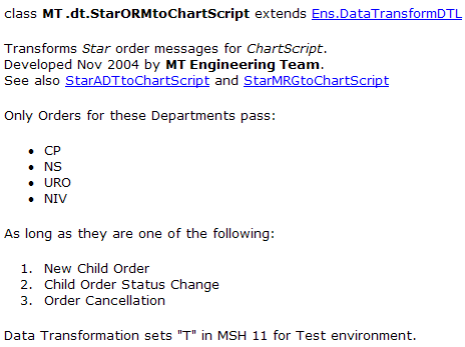
If you add Class Reference comments using the **Description** field with a Studio wizard or in the Studio Inspector window, Studio handles these details for you (apart from the length restriction). If you add Class Reference comments directly into the code, Studio alerts you to some Class Reference syntax errors: for example, if you insert a blank line between the comments and the declaration, or if you use an insufficient number of slashes at the beginning of a line within a Class Reference text block. However, Studio does not alert you to any other types of bad syntax within Class Reference comments.

Class Reference comments allow plain text, plus any standard HTML element and a small number of specialized elements, as shown in the following code sample:

### Class Definition

```
/// <p>Transforms <i>Star</i> order messages for <i>ChartScript</i>. <br/>
/// Developed Nov 2004 by <b>MT Engineering Team</b>. <br/>
/// See also <class>StarADTtoChartScript</class> and
/// <class>StarMRGtoChartScript</class> </p>
/// <p>Only Orders for these Departments pass: </p>
/// <ul><li>CP</li><li>NS</li><li>URO</li><li>NIV</li></ul>
/// <p>As long as they are one of the following:</p>
/// <ol>
/// <li>New Child Order</li>
/// <li>Child Order Status Change</li>
/// <li>Order Cancellation</li>
/// </ol>
/// <p>Data Transformation sets "T" in MSH 11 for Test environment.</p>
Class MT.dt.StarORMtoChartScript
      Extends Ens.DataTransformDTL [ ProcedureBlock ]

{
  // The data transformation class code goes here.
}
```

The previous example formats the *Class Reference* entry for the class as follows:



class **MT .dt.StarORMtoChartScript** extends Ens.DataTransformDTL

Transforms *Star* order messages for *ChartScript*.
Developed Nov 2004 by **MT Engineering Team**.
See also StarADTtoChartScript and StarMRGtoChartScript

Only Orders for these Departments pass:

- CP
- NS
- URO
- NIV

As long as they are one of the following:

1. New Child Order
2. Child Order Status Change
3. Order Cancellation

Data Transformation sets "T" in MSH 11 for Test environment.

## 2.8.3 Using HTML Markup in Class Documentation

You can use HTML tags within the comments in a class. With regard to the allowed HTML elements, adhere to as strict an HTML standard as you can, for example XHTML. This ensures that your comments can be interpreted by any browser. In addition to standard HTML, you can use the following tags: CLASS, METHOD, PROPERTY, PARAMETER, QUERY, and EXAMPLE. (As with standard HTML tags, the names of these tags are not case-sensitive.) The most commonly used tags are described here. See the documentation for %CSP.Documatic for details of the others.

### CLASS

Use to tag class names. If the class exists, the contents are displayed as a link to the class' documentation. For example:

```
/// This uses the <CLASS>Sample.Person</CLASS> class.
```

### EXAMPLE

Use to tag programming examples. This tag affects the appearance of the text. Note that each `///` line becomes a separate line in the example (in contrast to the usual case, where the lines are combined into a single paragraph). For example:

```
/// <EXAMPLE>
/// set o=..%New()
/// set o.MyProperty=42
/// set o.OtherProp="abc"
/// do o.WriteSummary()
/// </EXAMPLE>
```

### METHOD

Use to tag method names. If the method exists, the contents are displayed as a link to the method's documentation. For example:

```
/// This is identical to the <METHOD>Unique</METHOD> method.
```

### PROPERTY

Use to tag property names. If the property exists, the contents are displayed as a link to the property's documentation. For example:

```
/// This uses the value of the <PROPERTY>State</PROPERTY> property.
```

Here is a multi-line description using HTML markup:

```
/// The <METHOD>Factorial</METHOD> method returns the factorial
/// of the value specified by <VAR>x</VAR>.
```

# 2.9 Compiling Classes

Caché class definitions are compiled into application routines by the class compiler. Classes cannot be used in an application before they are compiled.

The class compiler differs from the compilers available with other programming languages, such as C++ or Java, in two significant ways: first, the results of compilation are placed into a shared repository (database), not a file system. Second, it automatically provides support for persistent classes.

Specifically, the class compiler does the following:

1. It generates a list of *dependencies* — classes that must be compiled first. Depending on the compile options used, any dependencies that have been modified since last being compiled will also be compiled.
2. It resolves *inheritance* — it determines which methods, properties, and other class members are inherited from superclasses. It stores this inheritance information into the class dictionary for later reference.
3. For persistent and serial classes, it determines the storage structure needed to store objects in the database and creates the necessary runtime information needed for the SQL representation of the class.
4. It executes any method generators defined (or inherited) by the class.
5. It creates one or more routines that contain the runtime code for the class. The class compiler groups methods according to language (ObjectScript and Basic) and generates separate routines, each containing methods of one language or the other.

   If you specify the **Keep Generated Source** option with the class compiler, you can view the source for the routines using the **View Other Code** command (from the **View** menu) within Studio.
6. It compiles all of the generated routines into executable code.
7. It creates a class descriptor. This is a special data structure (stored as a routine) that contains all the runtime dispatch information needed to support a class (names of properties, locations of methods, and so on).

## 2.9.1 Invoking the Class Compiler

There are several ways to invoke the class compiler:

- From within Studio using the option in the **Build** menu.
- From the Caché command line (in the Terminal) using the **Compile()** method of the %SYSTEM.OBJ object:

  **ObjectScript**

  ```
  Do $System.OBJ.Compile("MyApp.MyClass")
  ```

If you use SQL DDL statements to create a table, the class compiler is automatically invoked to compile the persistent class that corresponds to the table.

## 2.9.2 Class Compiler Notes

### 2.9.2.1 Compilation Order

When you compile a class, Caché also recompiles other classes if the class that you are compiling contains information about dependencies. For example, the system compiles any subclasses of the class. On some occasions, you may need to

control the order in which the classes are compiled. To do so, use the System, DependsOn, and CompileAfter keywords. For details, see the *Caché Class Definition Reference*.

To find the classes that the compiler will recompile when you compile a given class, use the **$SYSTEM.OBJ.GetDependencies()** method. For example:

```
SAMPLES>d $system.OBJ.GetDependencies("Sample.Address",.included)

SAMPLES>zw included
included("SOAP.Demo.LookupCity")=""
included("SOAP.DemoProxy.LookupCity")=""
included("Sample.Address")=""
included("Sample.Customer")=""
included("Sample.Employee")=""
included("Sample.Person")=""
included("Sample.Vendor")=""
```

The signature of this method is as follows:

```
classmethod GetDependencies(ByRef class As %String, Output included As %String, qspec As %String) as
%Status
```

Where:

- *class* is either a single class name (as in the example), a comma-separated list of class names, or a multidimensional array of class names. (If it is a multidimensional array, be sure to pass this argument by reference.) It can also include wildcards.
- *included* is a multidimensional array of the names of the classes that will be compiled when *class* is compiled.
- *qspec* is a string of compiler flags and qualifiers. See the next subsection. If you omit this, the method considers the current compiler flags and qualifiers.

### 2.9.2.2 Viewing Class Compiler Flags and Qualifiers

The **Compile()** method also allows you to supply flags and qualifiers that affect the result. Their position in the argument list is described in the explanation of the **Compile()** method. To view the applicable flags, execute the command:

#### ObjectScript

```
 Do $System.OBJ.ShowFlags()
```

This produces the following output:

```
    b - Include sub classes.
    c - Compile. Compile the class definition(s) after loading.
    d - Display. This flag is set by default.
    e - Delete extent.
    h - Generate help.
    i - Validate XML export format against schema on Load.
    k - Keep source.  When this flag is set, source code of
        generated routines will be kept.
    l - Lock classes while compiling.  This flag is set by default.
    p - Percent.  Include classes with names of the form %*.
    r - Recursive.  Compile all the classes that are dependency predecessors.
    s - Process system messages or application messages.
    u - Update only.  Skip compilation of classes that are already up-to-date.
    y - Include classes that are related to the current class in the way that
        they either reference to or are referenced by the current class in SQL usage.

These flags are deprecated a, f, g, o, q, v
Default flags for this namespace =dil
You may change the default flags with the SetFlags(flags,system) classmethod.
```

To view the full list of qualifiers, along with their description, type, and any associated values, execute the command:

**ObjectScript**

```
Do $System.OBJ.ShowQualifiers()
```

Qualifier information displays in a format similar to one of the following:

```
           Name: /checkschema
    Description: Validate imported XML files against the schema definition.
           Type: logical
           Flag: i
  Default Value: 1

           Name: /checksysutd
    Description: Check system classes for up-to-dateness
           Type: logical
  Default Value: 0

           Name: /checkuptodate
    Description: Skip classes or expanded classes that are up-to-date.
           Type: enum
           Flag: ll
      Enum List: none,all,expandedonly,0,1
  Default Value: expandedonly
  Present Value: all
  Negated Value: none
```

### 2.9.2.3 Compiling Classes that Include Bitmap Indices

When compiling a class that contains a bitmap index, the class compiler generates a bitmap extent index if no bitmap extent index is defined for that class. Special care is required when adding a bitmap index to a class on a production system. For more information, see the section "Generating a Bitmap Extent Index" in the "Defining and Building Indices" chapter of *Caché SQL Optimization Guide*.

### 2.9.2.4 Compiling When There Are Existing Instances of a Class in Memory

If the compiler is called while an instance of the class being compiled is open, there is no error. The already open instance continues to use its existing code. If another instance is opened after compilation, it uses the newly compiled code.

# 2.10 Making Classes Deployed

You might want to make some of your classes deployed before you send them to customers; this process hides the source code.

For any class definitions that contain method definitions that you do not want customers to see, compile the classes and then use **$SYSTEM.OBJ.MakeClassDeployed()**. For example:

**ObjectScript**

```
d $system.OBJ.MakeClassDeployed("MyApp.MyClass")
```

For an alternative approach, see the article *Adding Compiled Code to Customer Databases*.

## 2.10.1 About Deployed Mode

When a class is in deployed mode, its method and trigger definitions have been removed. (Note that if the class is a data type class, its method definitions are retained because they may be needed at runtime by cached queries.)

You can open the class definition in Studio, but it is read-only.

You cannot export or compile a deployed class, but you can compile its subclasses (if they are not deployed).

There is no way to reverse or undo deployment of a class. You can, however, replace the class by importing the definition from a file, if you previously exported it. (This is useful if you accidentally put one of your classes into deployed mode prematurely.)

# 3

# Package Options

This chapter discusses packages in more detail. Topics include:

- Overview
- Package names
- How to define packages
- How and why to define package mappings
- How to use packages when referring to classes
- How to import packages

When viewing this book online, use the preface of this book to quickly find other topics.

For persistent classes, the package is represented in SQL as an SQL schema. For details, see "Projection of Packages to Schemas," later in this book.

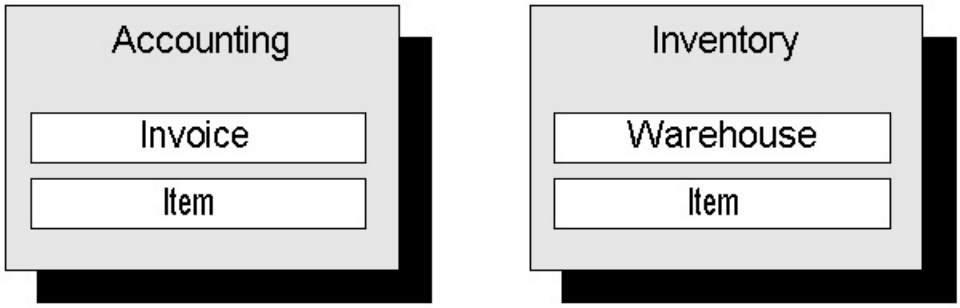
**Important:** When Caché encounters a reference to a class that does not include a package name and where the class name starts with "%", Caché assumes the class is in the "%Library" package.

## 3.1 Overview of Packages

Caché supports *packages*, which group related classes within a specific database. Packages provide the following benefits:

- They give developers an easier way to build larger applications and to share code with one another.
- They make it easier to avoid name conflicts between classes.
- They give a logical way to represent SQL schemas within the object dictionary in a clean, simple way: A package corresponds to a schema.

A package is simply a way to group related classes under a common name. For example, an application could have an "Accounting" system and an "Inventory" system. The classes that make up these applications could be organized into an "Accounting" package and an "Inventory" package:




Any of these classes can be referred to using their full name (which consists of package and class name):

**ObjectScript**

```
Do ##class(Accounting.Invoice).Method()
Do ##class(Inventory.Item).Method()
```

If the package name can be determined from context (see below), then the package name can be omitted:

**ObjectScript**

```
Do ##class(Invoice).Method()
```

As with classes, a package definition exists within a Caché database. For information on mapping a package from a database to a namespace, see the section "Package Mapping."

# 3.2 Package Names

A package name is a string. It may contain "." (period) characters, but no other punctuation. Each period-delimited piece of the package name is a subpackage, and there can be multiple subpackages. If you give a class the name Test.Subtest.TestClass, then this indicates that the name of the package is Test, the name of the subpackage is Subtest, and the name of the class is TestClass.

There are several limitations on the length and usage of package names:

- A package name is subject to a length limit. See "Classes" in Rules and Guidelines for Identifiers in the *Caché Programming Orientation Guide*.
- Within a namespace, each package name must be unique without regards to case. Hence, there cannot be both "ABC" and "abc" packages in a namespace, and the "abc.def" package and subpackage are treated as part of the "ABC" package.

For general information on identifiers, see the section "Naming Conventions" in the chapter "Defining and Compiling Classes."

# 3.3 Defining Packages

Packages are implied by the name of the classes. When you create a class, the package is automatically defined. Similarly, when the last class in a package is deleted, the package is also automatically deleted.

The following shows an example in which the package name is `Accounting`, the class name is `Invoice`, and the fully qualified class name is `Accounting.Invoice`:

### Class Definition

```
Class Accounting.Invoice
{
}
```

# 3.4 Package Mapping

By definition, each package is part of a particular database. Frequently, each database is associated with a namespace, where the database and the namespace share a common name. This is the case for various system-supplied databases and namespaces, such as SAMPLES and USER. To make a package definition in a database available to a namespace not associated with that database, use *package mapping*. This procedure is described in more detail in the in the *Caché System Administration Guide*; the following is an introduction. The database containing the package is the *source database* and the namespace into which the package is being mapped as the *target namespace.* To map a package, the procedure is:

1. From the Management Portal home page, go to the **Namespaces** page (**System Administration** > **Configuration** > **System Configuration** > **Namespaces**).
2. On the **Namespaces** page, select the target namespace by clicking **Package Mappings** on that corresponding row in the table. This displays the **Package Mappings** page for the target namespace.
3. On the **Package Mappings** page, click **New**. This displays a dialog for setting up the mapping.
4. In this dialog, complete the fields as follows:
   - **Package Database Location** — The source database.
   - **Package Name** — The package being mapped. If you plan to map a package that has not yet been created, you can specify its name in advance by clicking **Specify a New Package** and entering the name of the package.

   Click **OK** to use these values and dismiss the dialog.
5. The **Package Mappings** page should now display the mapping. Click **Save Changes** to save the mapping.

Mapping a package across namespace maps the package definition, not its data. Hence, mapping the Sample package from the SAMPLES namespace to the USER namespace does not make the instances of the Sample.Person from the SAMPLES namespace available in the USER namespace.

**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 namespace. See "Add Global, Routine, and Package Mapping to a Namespace" in the chapter "Configuring Caché" in the *Caché System Administration Guide*.

When you map a package, the mapping applies to the class definitions in that package *and* to the generated routines, which are in the same package.

### 3.4.1 Mapping a Package Across Multiple Namespaces

Caché also provides functionality to make a source package available in multiple target namespaces through a single action. Such a mapping makes the package available in all namespaces except DOCBOOK and SAMPLES.

To make a package available to multiple namespaces, the procedure is:

1. Create a namespace called %ALL according the instructions in the "Create a Namespace" section of the "Configuring Caché" chapter of the *Caché System Administration Guide*.
2. Create a package mapping as described in this section and save it.

The classes in the mapped package are then visible and usable in the %SYS namespace, the USER namespace, and any user-defined namespaces.

**Note:** Deleting the %ALL namespace removes its mappings.

## 3.5 Package Use When Referring to Classes

There are two ways to refer to classes:

- Use the fully qualified name (that is, *Package*.*Class*). For example:

  **ObjectScript**

  ```
  // create an instance of Lab.Patient
  Set patient = ##class(Lab.Patient).%New()
  ```

- Use the short class name and let the class compiler resolve which package it belongs to.

  By default, when you use a short class name, Caché assumes that the class is either in the package of the class whose code you are using (if any), or in the %Library package, or in the User package.

  If you want the compiler to search for classes in other packages, import those packages as described in the next section.

  **Note:** It is an error to use a short class that is ambiguous; that is, if you have the same short class name in two or more packages and import all of them, you will get an error when the compiler attempts to resolve the package name. To avoid this error, use full names.

## 3.6 Importing Packages

When you import packages, Caché looks for any short class names in those packages. In a class definition, you can import a package via the class Import directive or the ObjectScript #IMPORT directive. This section explains these directives, discusses the effect on the User package and the effect on subclasses, and presents some tips.

## 3.6.1 Class Import Directive

You can include the class Import directive at the top of a class definition, before the `Class` line. The syntax for this directive is as follows:

```
Import packages
Class name {}
```

Where *packages* is either a single package or a comma-separated list of packages, enclosed in parentheses. The word `Import` is not case-sensitive, but is usually capitalized as shown here.

Remember that in a class context, the current package is always implicitly imported.

## 3.6.2 ObjectScript #IMPORT Directive

In ObjectScript method, an #IMPORT directive imports a package so that you can use short class names to refer to classes in it. The syntax for this directive is as follows:

```
#import packagename
```

Where *packagename* is the name of the package. The word `#import` is not case-sensitive. For example:

**ObjectScript**

```
#import Lab
 // Next line will use %New method of Lab.Patient, if that exists
 Set patient = ##class(Patient).%New()
```

You can have multiple #IMPORT directives:

**ObjectScript**

```
#import Lab
#import Accounting

 // Look for "Patient" within Lab & Accounting packages.
 Set pat = ##class(Patient).%New()

 // Look for "Invoice" within Lab & Accounting packages.
 Set inv = ##class(Invoice).%New()
```

The order of #IMPORT directives has no effect on how the compiler resolves short class names.

## 3.6.3 Explicit Package Import Affects Access to User Package

Once your code imports any packages explicitly, the User package is *not* automatically imported. If you need that package, you must import it explicitly as well. For example:

**ObjectScript**

```
#import MyPackage
#import User
```

The reason for this logic is because there are cases where you may *not* want the User package to be imported.

## 3.6.4 Package Import and Inheritance

A class inherits any explicitly imported packages from the superclasses.

The name of a class is resolved in the context where it was first used and not with the current class name. For example, suppose you define in User.MyClass a method **MyMethod()** and then you create a MyPackage.MyClass class that inherits from User.MyClass and compile this. Caché compiles the inherited **MyMethod()** method in MyPackage.MyClass — but resolves any class names in this method in the context of User.MyClass (because this is where this method was defined).

## 3.6.5 Tips for Importing Packages

By importing packages, you can make more adaptable code. For example, you can create code such as:

### ObjectScript

```
#import Customer1
 Do ##class(Application).Run()
```

Now change App.MAC to:

### ObjectScript

```
#import Customer2
 Do ##class(Application).Run()
```

When you recompile App.MAC, you will be using the Customer2.Application class. Such code requires planning: you have to consider code compatibility as well as the effects on your storage structures.

# 4

# Defining and Referring to Class Parameters

This chapter describes how to define class parameters and how to refer to them programmatically. It discusses the following topics:

- Introduction
- How to define class parameters
- Parameter types and values
- How to refer to parameters of a class

When viewing this book online, use the preface of this book to quickly find other topics.

## 4.1 Introduction to Class Parameters

A class parameter defines a special constant value available to all objects of a given class. When you create a class definition (or at any point before compilation), you can set the values for its class parameters. By default, the value of each parameter is the null string, but you can specify a non-null value as part of the parameter definition. At compile-time, the value of the parameter is established for all instances of a class. With rare exceptions, this value cannot be altered at runtime.

Class parameters are typically used for the following purposes:

- To customize the behavior of the various data type classes (such as providing validation information).
- 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 define a class-specific constant value.
- To provide parameterized values for method generator methods to use. A method generator is a special type of method whose implementation is actually a short program that is run at compile-time in order to generate the actual runtime code for the method. Many method generators use class parameters.

# 4.2 Defining Class Parameters

To add a class parameter to a class definition, add an element like one of the following to the class:

```
Parameter PARAMNAME;
Parameter PARAMNAME as Type;
Parameter PARAMNAME as Type = value;
Parameter PARAMNAME as Type [ Keywords ] = value;
```

Where

- *PARAMNAME* is the name of the parameter. Note that by convention, parameters in Caché system classes are nearly all in uppercase; this convention provides an easy way to distinguish parameters from other class members, merely by name. There is no requirement for you to do the same.
- *Type* is a parameter type. See the next section.
- *value* is the value of the parameter. In most cases, this is a literal value such as 100 or `"MyValue"`. For some types, this can be an ObjectScript expression. See the next section.
- *Keywords* represents any parameter keywords. These are optional. For an introduction to keywords, see "Compiler Keywords," earlier in this book. For parameter keywords; see "Parameter Keywords" in the *Caché Class Definition Reference*.

# 4.3 Parameter Types and Values

It is not necessary to specify a parameter type, but if you do, this information is primarily meant for use by the development environment. The Class Inspector in Studio uses the parameter type to provide suitable options and validation. For example, if a parameter is of type BOOLEAN, the Class Inspector provides the choices 0 and 1.

The parameter types include BOOLEAN, STRING, and INTEGER. Note that these are not Caché class names. For a complete list, see "Parameter Definitions" in the *Caché Class Definition Reference*.

Except for the types COSEXPRESSION and CONFIGVALUE (both described in subsections), the compiler ignores the parameter types.

## 4.3.1 Class Parameter to Be Evaluated at Runtime (COSEXPRESSION)

You can define a parameter as an ObjectScript expression that it is evaluated at runtime. To do so, specify its type as COSEXPRESSION and specify an ObjectScript expression as the value:

### Class Member

```
Parameter PARAMNAME As COSEXPRESSION = "ObjectScriptExpression";
```

where *PARAMNAME* is the parameter being defined and *ObjectScriptExpression* is the ObjectScript content that is evaluated at runtime.

An example class parameter definition would be:

#### Class Member

```
Parameter DateParam As COSEXPRESSION = "$H";
```

### 4.3.2 Class Parameter to Be Evaluated at Compile Time (Curly Braces)

You can define a parameter as an ObjectScript expression that it is evaluated at compile time. To do so, specify no type and specify the value in curly braces:

#### Class Member

```
Parameter PARAMNAME = {ObjectScriptExpression};
```

where *PARAMNAME* is the parameter being defined and *ObjectScriptExpression* is the ObjectScript content that is evaluated at runtime.

For example:

#### Class Member

```
Parameter COMPILETIME = {$zdatetime($h)};
```

### 4.3.3 Class Parameter to Be Updated at Runtime (CONFIGVALUE)

You can define a parameter so that it can modified outside of the class definition. To do so, specify its type as CONFIGVALUE. In this case, you can modify the parameter via the **$SYSTEM.OBJ.UpdateConfigParam()** method. For example, the following changes the value of the parameter *MYPARM* (in the class MyApp.MyClass) so that its new value is 42:

```
set sc=$system.OBJ.UpdateConfigParam("MyApp.MyClass","MYPARM",42)
```

Note that **$SYSTEM.OBJ.UpdateConfigParam()** affects the generated class descriptor as used by any new processes, but does not affect the class definition. If you recompile the class, Caché regenerates the class descriptor, which will now use the value of this parameter as contained in the class definition, thus overwriting the change made via **$SYSTEM.OBJ.UpdateConfigParam()**.

## 4.4 Referring to Parameters of a Class

To refer to a parameter of a class, you can do any of the following:

- Within a method of the associated class, use the following expression:

  ```
  ..#PARMNAME
  ```

  You can use this expression with the DO and SET commands, or you can use it as part of another expression. The following shows one possibility:

  ```
  set object.PropName=..#PARMNAME
  ```

  In the next variation, a method in the class checks the value of a parameter and uses that to control subsequent processing:

  ```
  if ..#PARMNAME=1 {
    //do something
  } else {
    //do something else
  }
  ```

- To access a parameter in any class, use the following expression:

  ```
  ##class(Package.Class).#PARMNAME
  ```

  where *Package.Class* is the name of the class and *PARMNAME* is the name of the parameter.

  This syntax accesses the given class parameter and returns its value. You can use this expression with commands such as DO and SET, or you can use it as part of another expression. The following shows an example:

  **ObjectScript**

  ```
  w ##class(%XML.Adaptor).#XMLENABLED
  ```

  displays whether methods generated by the XML adaptor are XML enabled, which by default is set to 1.

- To access the parameter, where the parameter name is not determined until runtime, use the $PARAMETER function:

  ```
  $PARAMETER(classnameOrOref,parameter)
  ```

  where *classnameOrOref* is either the fully qualified name of a class or an OREF of an instance of the class, and *parameter* evaluates to the name of a parameter in the associated class.

  For information on OREFs, see "Working with Registered Objects."

  For more information, see the $PARAMETER page in the *Caché ObjectScript Reference*.

# 5

# Defining and Calling Methods

This chapter describes the rules and options for creating methods in Caché classes and for calling those methods. It discusses the following topics:

- Introduction to methods
- How to define methods
- How to specify method arguments
- How to indicate how arguments are to be passed
- How to specify a variable number of arguments
- How to return a value
- How to specify the implementation language
- Types of methods (CodeMode keyword)
- How to project a method as a stored procedure
- How to call class methods
- How to cast a method
- How to override an inherited method

For information on calling instance methods, see the next chapter; such methods apply only to object classes.

When viewing this book online, use the preface of this book to quickly find other topics.

## 5.1 Introduction to Methods

A method is an executable element defined by a class. Caché supports two types of methods: instance methods and class methods. An *instance method* is invoked from a specific instance of a class and typically performs some action related to that instance. A *class method* is a method that can be invoked without reference to any object instance; these are called static methods in other languages.

The term *method* usually refers to an instance method. The more specific phrase *class method* is used to refer to class methods.

Because you cannot execute an instance method without an instance of an object, instance methods are useful only when defined in object classes. In contrast, you can define class methods in any kind of class.

# 5.2 Defining Methods

To add a class method to a class, add an element like the following to the class definition:

```
ClassMethod MethodName(Arguments) as Classname [ Keywords]
{
//method implementation
}
```

Where:

- *MethodName* is the name of the method. For rules, see "Naming Conventions," earlier in this book.
- *Arguments* is a comma-separated list of arguments. For details, see "Specifying Method Arguments."
- *Classname* is an optional class name that represents the type of value (if any) returned by this method. Omit the `As` *Classname* part if the method does not return a value.

  The class can be a data type class, an object class, or (less commonly) a class of no type. The class name can be a complete class name or a short class name. For details, see "Package Use When Referring to Classes," in the chapter "Package Options."
- *Keywords* represents any method keywords. These are optional. See "Compiler Keywords," earlier in this book. Later sections of this chapter discuss additional keywords.
- The method implementation depends on the implementation language and type of method; see "Specifying the Implementation Language" and "Types of Methods." By default, the method implementation consists of zero or more lines of ObjectScript.

To add an instance method to a class, use the same syntax with `Method` instead of `ClassMethod`:

```
Method MethodName(arguments) as Classname [ Keywords]
{
 //method implementation
}
```

Instance methods are relevant only in object classes.

# 5.3 Specifying Method Arguments: Basics

A method can take any number of arguments. The method definition must specify the arguments that it takes. It can also specify the type and default value for each argument. (In this context, type refers to any kind of class, not specifically data type classes.)

Consider the following generic class method definition:

```
ClassMethod MethodName(Arguments) as Classname [ Keywords]
{
 //method implementation
}
```

Here *Arguments* has the following general form:

```
argname1 as type1 = value1, argname2 as type2 = value2, argname3 as type3 = value3, [and so on]
```

Where:

- *argname1*, *argname2*, *argname3*, and so on are the names of the arguments. These names must follow the rules for variable names.
- *type1*, *type2*, *type3*, and so on are class names. This part of the method definition is intended to describe, to programmers who might use this method, what type of value to pass for the corresponding argument. Generally it is a good idea to explicitly specify the type of each method argument.

  Typically the types are data type classes or object classes.

  The class name can be a complete class name or a short class name. For details, see "Package Use When Referring to Classes," in the chapter "Defining and Using Packages."

  You can omit this part of the syntax. If you do, also omit the `as` part.
- *value1*, *value2*, *value3*, and so on are the default values of the arguments. The method automatically sets the argument equal to this value if the method is called without providing a value for the argument.

  Each value can either be a literal value (`"abc"` or 42) or an ObjectScript expression enclosed in curly braces. For example:

  **Class Member**

  ```
  ClassMethod Test(flag As %Integer = 0)
  {
   //method implementation
  }
  ```

  For another example:

  **Class Member**

  ```
  ClassMethod Test(time As %Integer = {$horolog} )
  {
   //method implementation
  }
  ```

  You can omit this part of the syntax. If you do, also omit the equals sign (=).

For instance, here is a **Calculate()** method with three arguments:

**Class Member**

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

where *count* and *state* are declared as %Integer and %String, respectively. By default, arguments are of the %String data type, so that an argument of unspecified type is a %String. This is the case for *name* in the above example.

# 5.4 Indicating How Arguments Are to Be Passed

The method definition also indicates, to programmers who might use the method, how each argument is expected to be passed. Arguments can be passed by value (the default technique) or by reference. See "Passing Arguments to a Method" later in this chapter.

It may or may not be sensible to pass a specific argument by reference. The details depend upon the method implementation. Consequently, when you define a method, you should use the method signature to indicate to other programmers how each argument is meant to be used.

To indicate that an argument *should* be passed by reference, include the ByRef modifier in the method signature, before the name of the argument. The following shows an example that uses ByRef for both its arguments:

### Class Member

```
/// Swap value of two integers
Method Swap(ByRef x As %Integer, ByRef y As %Integer)
{
    Set temp = x
    Set x = y
    Set y = temp
}
```

Similarly, to indicate that an argument *should* be passed by reference *and* is intended to have no incoming value, include the Output modifier in the method signature, before the name of the argument. For example:

### Class Member

```
Method CreateObject(Output newobj As MyApp.MyClass) As %Status
{
    Set newobj = ##class(MyApp.MyClass).%New()
    Quit $$$OK
}
```

# 5.5 Specifying a Variable Number of Arguments

You can define a method that accepts variable numbers of arguments. To do so, include `...` after the name of the last argument, as in the following example. This example also shows how this feature can be used.

### Class Member

```
ClassMethod MultiArg(Arg1... As %String)
{
 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
}
```

The following Terminal session shows how this method behaves:

```
SAMPLES>do ##class(VarNumArg.Demo).MultiArg("scooby","shaggy","velma","daphne","fred")
Invocation has 5 elements
Argument[1]:   scooby
Argument[2]:   shaggy
Argument[3]:   velma
Argument[4]:   daphne
Argument[5]:   fred
```

For more details on this feature, see the "Variable Numbers of Arguments" section of the "User-defined Code" chapter of *Using Caché ObjectScript*.

# 5.6 Returning a Value

To define a method so that it returns a value, use either of the following in the method (if you implement the method in ObjectScript):

```
Return returnvalue
```

Or:

```
Quit returnvalue
```

Where *returnvalue* is a suitable value for the method to return. This should be consistent with the declared return type of the method. If the return type is a data type class, the method should return a literal value. If the return type is an object class, the method should return an instance of that class (specifically an OREF). For details, see the chapter "Working with Registered Objects."

For example:

### Class Member

```
ClassMethod Square(input As %Numeric) As %Numeric
{
    Set returnvalue = input * input
    Return person
}
```

For another example, this method returns an object instance:

### Class Member

```
ClassMethod FindPerson(id As %String) As Person
{
    Set person = ##class(Person).%OpenId(id)
    Return person
}
```

The syntax for returning a value is different depending on the implementation language of the method.

# 5.7 Specifying the Implementation Language

You have the choice of implementation language when creating a method. In fact, within a single class, it is possible to have multiple methods implemented in different languages. All methods interoperate regardless of implementation language.

By default, a method uses the language specified by the Language keyword of the class to which it belongs. For this keyword, the default is `cache` (ObjectScript). The other options are `basic` (Caché Basic), `java` (Java), `javascript` (JavaScript), `mvbasic` (MVBasic), and `tsql` (TSQL).

You can override this for a specific method by setting the Language keyword for that method:

### Class Definition

```
Class MyApp.Test {

/// A Basic method
Method TestB() As %Integer [ Language = basic]
{
    'This is Basic
    Print "This is a test"
    Return 1
}

/// An ObjectScript method
Method TestC() As %Integer [ Language = cache]
{
    // This is ObjectScript
    Write "This is a test"
    Quit 1
}
}
```

# 5.8 Types of Methods (CodeMode Options)

Caché supports four types of methods, which the class compiler handles differently:

- Code methods (the most default and the most common)
- Expression methods
- Call methods
- Method generators

## 5.8.1 Code Methods

A code method is a method whose implementation is simply lines of code. This is the most typical type of method and is the default.

The method implementation can contain any code that is valid for the implementation language.

**Note:** Caché comes with a set of system-defined methods that perform simple, common tasks. If a user-defined method performs one of these tasks, then the compiler does not generate any executable code for it. Rather, it associates the user-defined method with the system-defined method, so that invoking the user-defined method results in a call to the system-defined method, with an associated performance benefit. Also, the debugger does not step through such a system-defined method.

## 5.8.2 Expression Methods

An expression method is a method that may be replaced by the class compiler, in certain circumstances, with a direct in-line substitution of a specified expression. Expression methods are typically used for simple methods (such as those found in data type classes) that need rapid execution speed.

For example, it is possible to convert the **Speak()** method of the Dog class from the previous example into an expression method:

### Class Member

```
Method Speak() As %String [CodeMode = expression]
{
    "Woof, Woof"
}
```

Assuming *dog* refers to a Dog object, this method could be used as follows:

**ObjectScript**

```
Write dog.Speak()
```

Which could result in the following code being generated:

**ObjectScript**

```
Write "Woof, Woof"
```

It is a good idea to give default values to all formal variables of an expression method. This prevents potential inline substitution problems caused by missing actual variables at runtime.

**Note:** Caché does not allow macros or call-by-reference arguments within expression methods.

### 5.8.3 Call Methods

A call method is a special mechanism to create method wrappers around existing Caché routines. This is typically useful when migrating legacy code to object-based applications.

Defining a method as a call method indicates that a specified routine is called whenever the method is invoked. The syntax for a call method is:

**Class Member**

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

where "Tag^Routine" specifies a tag name within a routine.

### 5.8.4 Method Generators

A method generator is actually a small 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, see "Defining Method and Trigger Generators."

## 5.9 Projecting a Method As an SQL Stored Procedure

You can define a class method (but not an instance method) so that it is also available as an SQL stored procedure. To do so, include the SqlProc keyword in the method definition.

The default name of the procedure is as *CLASSNAME_METHODNAME* To specify a different name, specify the SqlName keyword.

For details, see "Defining Stored Procedures" in *Using Caché SQL*.

# 5.10 Calling Class Methods

This section discusses how to call class methods in ObjectScript. This section applies to all kinds of classes. Note that instance methods are discussed in the next chapter, because they apply only to object classes.

- To call a class method of any class (if that method is not private), use the following expression:

  ```
  ##class(Package.Class).Method(Args)
  ```

  Where *Package.Class* is the name of the class, *Method* is the name of the method, and *Args* is any arguments of the method.

  `##class` is not case-sensitive.

  This expression invokes the given class method and obtains its return value (if any). You can use this expression with commands such as DO and SET, or you can use it as part of another expression. The following shows some variations:

  **ObjectScript**

  ```
  do ##class(Package.Class).Method(Args)
  set myval= ##class(Package.Class).Method(Args)
  write ##class(Package.Class).Method(Args)
  set newval=##class(Package.Class).Method(Args)_##class(Package2.Class2).Method2(Args)
  ```

  You can omit the package. If you do so, the class compiler determines the correct package name to use (this name resolution is explained in the "Packages" chapter).

- (Within a class method) to call another class method of that class (which could be an inherited method), use the following expression:

  ```
  ..MethodName(args)
  ```

  You can use this expression with the DO command. If the method returns a value, you can use SET, or you can use it as part of another expression. The following shows some variations:

  ```
  do ..MethodName()
  set value=..MethodName(args)
  ```

  **Note:** You cannot use this syntax in a class method to refer to a property or an instance method, because those references require the instance context.

- To execute a class method, where the method name is not determined until runtime, use the $CLASSMETHOD function:

  ```
  $CLASSMETHOD(classname, methodname, Arg1, Arg2, Arg3, ... )
  ```

  where *classname* evaluates to the fully qualified name of a class, *methodname* evaluates to the name of a class method in that class, and *Arg1*, *Arg2*, *Arg3*, and so on are any arguments to that method. For example:

  **ObjectScript**

  ```
  set cls="Sample.Person"
  set clsmeth="PrintPersons"
  do $CLASSMETHOD(cls,clsmeth)
  ```

  For more information, see the $CLASSMETHOD page in the *Caché ObjectScript Reference*.

If the given method does not exist or if it is an instance method instead, the system generates the <METHOD DOES NOT EXIST> error. If the given method is private, the system generates the <PRIVATE METHOD> error.

### 5.10.1 Passing Arguments to a Method

The default way to pass arguments to methods is by value. In this technique, simply include the arguments as variables, literal values, or other expressions within the method call, as shown in the preceding examples.

It is also possible to pass arguments by reference.

This works follows: the system 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 to a method, 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. You can pass the memory address instead; this technique is known as *calling by reference*. It is also the only way to pass a multidimensional array as an argument.

In ObjectScript, to pass an argument by reference, precede that argument with a period. For example:

#### ObjectScript

```
 set MyArg(1)="value A"
 set MyArg(2)="value B"
 set status=##class(MyPackage.MyClass).MyMethod(.MyArg)
```

In this example, we pass a value (a multidimensional array) by reference so that the method could receive the value. In other cases, it is useful to pass an argument by reference so that you can use its value after running the method. For example:

```
 set status=##class(MyPackage.MyClass).GetList(.list)
 //use the list variable in subsequent logic
```

In other cases, you might specify a value for the variable, invoke a method that modifies it (and that returns it by reference), and then use the changed value.

## 5.11 Casting a Method

To cast a method of one class as a method of another class, the syntax is either of the following (in ObjectScript):

```
Do ##class(Package.Class1)Class2Instance.Method(Args)
Set localname = ##class(Package.Class1)Class2Instance.Method(Args)
```

You can cast both class methods and instance methods.

For example, suppose that two classes, MyClass.Up and MyClass.Down, both have **Go()** methods. For **MyClass.Up**, this method is as follows

#### Class Member

```
Method Go()
{
    Write "Go up.",!
}
```

For **MyClass.Down**, the **Go()** method is as follows:

#### Class Member

```
Method Go()
{
    Write "Go down.",!
}
```

You can then create an instance of MyClass.Up and use it to invoke the **MyClass.Down.Go** method:

```
>Set LocalInstance = ##class(MyClass.Up).%New()

>Do ##class(MyClass.Down)LocalInstance.Go()
Go down.
```

It is also valid to use ##class as part of an expression, as in

**ObjectScript**

```
 Write ##class(Class).Method(args)*2
```

without setting a variable equal to the return value.

A more generic approach is to use the $METHOD and $CLASSMETHOD functions, which are for instance and class methods, respectively. These are described in earlier sections of this chapter.

# 5.12 Overriding an Inherited Method

A class inherits methods (both class and instance methods) from its superclass or superclasses. Except for methods that are marked Final, you can override these definitions by providing a definition within this class. If you do so, note the following rules:

- If the method is a class method in the superclass, you cannot override it as an instance method in the subclass, and vice versa.
- The return type of the subclass method must be either the same as the original return type or a subclass of the original return type.
- The method in the subclass *can* have more arguments than the method in the superclass. (Also see the "Number of Arguments" subsection.)
- The method in the subclass can specify different default values for the arguments.
- The types of the arguments in the subclass method must be consistent with the types of the arguments in the original method. Specifically, any given argument must be either the same as the original type or a subclass of the original type.

  Note that if an argument has no specified type, the compiler treats the argument as %String. Thus if an argument in the superclass method has no type, the corresponding argument of a subclass method can be %String, can be a subclass of %String, or can have no type.
- The method in the subclass should receive argument values in the same way as the method in the superclass. For example, if a given argument is passed by reference in the superclass, the same argument should be passed by reference in the subclass.

  If the method signatures are inconsistent in this regard, it is harder for other developers to know how to use the methods appropriately. Note, however, that the compiler does not issue an error.

If your method implementation needs to call the method of the same name as defined in the superclass, you can use the syntax ##super(), which is discussed in the subsections. This discussion applies to code that is written in ObjectScript.

## 5.12.1 ##super()

Within a method, use the following expression to call the method of the same name as defined in the nearest superclass:

```
##super()
```

You can use this expression with the DO command. If the method returns a value, you can use SET, or you can use it as part of another expression. The following shows some variations:

```
do ##super()
set returnvalue=##super()_"additional string"
```

**Note:** *##super* is not case-sensitive. Also note that, unlike other features in this chapter, *##super()* is available within Basic methods as well as within ObjectScript methods.

This is useful if you define a method that should invoke the existing method of the superclass and then perform some additional steps such as modifying its returned value.

## 5.12.2 ##super and Method Arguments

*##super* also works with methods that accept arguments. If the subclass method does not specify a default value for an argument, make sure that the method passes the argument by reference to the superclass.

For example, suppose the code for the method in the superclass (**MyClass.Up.SelfAdd()**) is:

### Class Member

```
ClassMethod SelfAdd(Arg As %Integer)
{
    Write Arg,!
    Write Arg + Arg
}
```

then its output is:

```
>Do ##Class(MyClass.Up).SelfAdd(2)
2
4
>
```

The method in the subclass (**MyClass.Down.SelfAdd()**) uses *##super* and passes the argument by reference:

### Class Member

```
ClassMethod SelfAdd(Arg1 As %Integer)
{
    Do ##super(.Arg1)
    Write !
    Write Arg1 + Arg1 + Arg1
}
```

then its output is:

```
>Do ##Class(MyClass.Down).SelfAdd(2)
2
4
6
>
```

In **MyClass.Down.SelfAdd()**, notice the period before the argument name. If we omitted this and we invoked the method without providing an argument, we would receive an `<UNDEFINED>` error.

## 5.12.3 Calls That ##super Affects

*##super* only affects the current method call. If that method makes any other calls, those calls are relative to the current object or class, *not* the superclass. For example, suppose that MyClass.Up has **MyName()** and **CallMyName()** methods:

```
Class MyClass.Up Extends %Persistent
{

ClassMethod CallMyName()
{
    Do ..MyName()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Up",!
}

}
```

and that MyClass.Down overrides those methods as follows:

```
Class MyClass.Down Extends MyClass.Up
{

ClassMethod CallMyName()
{
    Do ##super()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Down",!
}

}
```

then invoking the **CallMyName()** methods have the following results:

```
USER>d ##class(MyClass.Up).CallMyName()
Called from MyClass.Up

USER>d ##class(MyClass.Down).CallMyName()
Called from MyClass.Down
```

**MyClass.Down.CallMyName()** has different output from **MyClass.Up.CallMyName()** because its **CallMyName()** method includes *##super* and so calls the **MyClass.Up.CallMyName()** method, which then calls the uncast **MyClass.Down.MyName()** method.

## 5.12.4 Number of Arguments

In some cases, you might find it necessary to add new arguments to a method in a superclass, thus resulting in more arguments than are defined in the method in a subclass. The subclasses will still compile, because (for convenience) the compiler appends the added arguments to the method in the subclass. In most cases, you should still examine all the subclasses that extend the method, and edit the signatures to account for the additional arguments, and decide whether you want to edit the code also. Even if you do not want to edit signatures or code, you still must consider two points:

- Make sure that the added argument names are not the same as the names of any variables used in the method in the subclass. The compiler appends the added arguments to the method in the subclass. If these arguments happen to have the same names as variables used in the method of the subclass, unintended results will occur.
- If the method in the subclass uses the added arguments (because this method uses *##super*), make sure that the method in the superclass specifies default values for the added arguments.

# 6

# Working with Registered Objects

The %RegisteredObject class is the basic object API in Caché. This chapter describes how to use this API. Information in this chapter applies to all subclasses of %RegisteredObject.

- Introduction
- OREF basics
- How to create new objects
- How to view object contents
- Introduction to dot syntax
- How to validate an object
- How to determine an object type
- How to clone objects
- How to refer to properties of an instance
- How to call methods of an instance
- How to obtain the class name from an instance
- $this variable (current instance)
- i%PropertyName (instance variables)

Also see the chapter "Working with Persistent Objects."

When viewing this book online, use the preface of this book to quickly find other topics.

## 6.1 Introduction to Object Classes

An *object class* is any class that inherits from %RegisteredObject. With an object class, you can do the following things:

- Create instances of the class. These instances are known as objects.
- Set properties of those objects.
- Invoke methods of those objects (instance methods).

These tasks are possible only with object classes.

The classes %Persistent and %SerialObject are subclasses of %RegisteredObject. These classes are described in later chapters. Also, for an overview, see "Object Classes" in the chapter "Defining and Compiling Classes."

# 6.2 OREF Basics

When you create an object, the system creates an in-memory structure, which holds information about that object, and also creates an OREF (object reference), which is a pointer to that structure.

The object classes provide several methods that create OREFs. When you work with any of the object classes, you use OREFs extensively. You use them when you specify values of properties of an object, access values of properties of the object, and call instance methods of the object. Consider the following example:

```
SAMPLES>set person=##class(Sample.Person).%New()

SAMPLES>set person.Name="Carter,Jacob N."

SAMPLES>do person.PrintPerson()

Name: Carter,Jacob N.
```

In the first step, we call the **%New()** method of the Sample.Person class, which creates an object and returns an OREF that points to that object. We set the variable `person` equal to this OREF. In the next step, we set the Name property of object. In the third step, we invoke the **PrintPerson()** instance method of the object. (Note that the Name property and the **PrintPerson()** method are both just examples—these are defined in the Sample.Person class but are not part of the general object interface.)

An OREF is transient; the value exists only while the object is in memory and is not guaranteed to be constant over different invocations.

**CAUTION:** An OREF is only valid within the namespace where it was created; hence, if there are existing OREFs and the current namespace changes, any OREF from the previous namespace is no longer valid. If you attempt to use OREFs from other namespaces, there might not be an immediate error, but the results cannot be considered valid or usable, and may cause disastrous results in the current namespace.

## 6.2.1 INVALID OREF Error

In simple expressions, if you try to set a property, access a property, or invoke an instance method of a variable that is not an OREF, you receive an `<INVALID OREF>` error. For example:

```
SAMPLES>write p2.PrintPerson()

WRITE p2.PrintPerson()
^
<INVALID OREF>
SAMPLES>set p2.Name="Dixby,Jase"

SET p2.Name="Dixby,Jase"
^
<INVALID OREF>
```

**Note:** The details are different when the expression has a chain of OREFs; see "Introduction to Dot Syntax."

## 6.2.2 Testing an OREF

Caché provides a function, **$ISOBJECT**, which you can use to test whether a given variable holds an OREF. This function returns 1 if the variable contains an OREF and returns 0 otherwise. If there is an chance that a given variable might not

contain an OREF, it is good practice to use this function before trying to set a property, access a property, or invoke an instance method of the variable.

## 6.2.3 OREFs, Scope, and Memory

Any given OREF is a *pointer* to an in-memory object to which other OREFs might also point. That is, the OREF (which is a variable) is distinct from the in-memory object (although, in practice, the terms *OREF* and *object* are often used interchangeably).

Caché manages the in-memory structure automatically as follows. For each in-memory object, Caché maintains a *reference count* — the number of references to that object. Whenever you set a variable or object property to refer to a object, its reference count is automatically incremented. When a variable stops referring to an object (if it goes out of scope, is killed, or is set to a new value), the reference count for that object is decremented. When this count goes to 0, the object is automatically destroyed (removed from memory) and its **%OnClose()** method (if present) is called.

For example, consider the following method:

### Class Member

```
Method Test()
{
    Set person = ##class(Sample.Person).%OpenId(1)
    Set person = ##class(Sample.Person).%OpenId(2)
}
```

This method creates an instance of Sample.Person and places a reference to it into the variable *person*. Then it creates another instance of Sample.Person and replaces the value of *person* with a reference to it. At this point, the first object is no longer referred to and is destroyed. At the end of the method, *person* goes out of scope and the second object is destroyed.

## 6.2.4 Removing an OREF

If needed, to remove an OREF, use the KILL command:

```
kill OREF
```

Where *OREF* is a variable that contains an OREF. This command removes the variable. If there are no further references to the object, this command also removes the object from memory, as discussed earlier.

## 6.2.5 OREFs, the SET Command, and System Functions

For some system functions (for example, $Piece, $Extract, and $List), Caché supports an alternative syntax that you can use to modify an existing value. This syntax combines the function with the SET command as follows:

```
SET function_expression = value
```

Where *function_expression* is a call to the system function, with arguments, and *value* is a value. For example, the following statement sets the first part of the `colorlist` string equal to `"Magenta"`:

```
SET $PIECE(colorlist,",",1)="Magenta"
```

It is not supported to modify OREFs or their properties in this way.

# 6.3 Creating New Objects

To create a new instance of a given object class, use the class method **%New()** of that class. This method creates an object and returns an OREF. The following shows an example:

### ObjectScript

```
Set person = ##class(MyApp.Person).%New()
```

The **%New()** method accepts an argument, which is ignored by default. If present, this argument is passed to **%OnNew()** callback method of the class, if defined. If **%OnNew()** is defined, it can use the argument to initialize the newly created object in some way. For details, see "Implementing Callback Methods," later in this book.

If you have complex requirements that affect how you create new objects of given class, you can provide an alternative method to be used to create instances of that class. Such a method would call **%New()** and then would initialize properties of the object as needed. Such a method is sometimes called a *factory method*.

# 6.4 Viewing Object Contents

The WRITE command writes output of the following form for an OREF:

```
n@Classname
```

Where *Classname* is the name of the class, and *n* is an integer that indicates a specific instance of this class in memory. For example:

```
SAMPLES>write p
8@Sample.Person
```

If you use the ZWRITE command with an OREF, Caché displays more information about the associated object.

```
SAMPLES>zwrite p
p=<OBJECT REFERENCE>[8@Sample.Person]
+----------------- general information ---------------
|      oref value: 1
|      class name: Sample.Person
|           %%OID: $lb("3","Sample.Person")
| reference count: 2
+----------------- attribute values ------------------
|       %Concurrency = 1  <Set>
|                DOB = 33589
|               Name = "Clay,George O."
|                SSN = "480-57-8360"
+----------------- swizzled references ---------------
|   i%FavoriteColors = ""  <Set>
|   r%FavoriteColors = ""  <Set>
|              i%Home = $lb("5845 Washington Blvd","St Louis","NM",55683)  <Set>
|              r%Home = ""  <Set>
|           i%Office = $lb("3413 Elm Place","Pueblo","WI",98532)  <Set>
|           r%Office = ""  <Set>
|           i%Spouse = ""
|           r%Spouse = ""
+-------------------------------------------------
```

Notice that this information displays the class name, the OID, the reference count, and the current values (in memory) of properties of the object. In the section `swizzled references`, the items with names starting `i%` are instance variables, which are discussed later in this chapter. (The items with names starting `r%` are for internal use only.)

# 6.5 Introduction to Dot Syntax

With an OREF, you can use *dot syntax* to refer to properties and methods of the associated object. This section introduces dot syntax, which is also discussed in later sections, along with alternative ways to refer to properties and methods of objects.

The general form of dot syntax is as follows:

```
oref.membername
```

For example, to specify the value of a property for an object, you can use a statement like this:

**ObjectScript**

```
Set oref.PropertyName = value
```

where *oref* is the OREF of the specific object, *PropertyName* is the name of the property that you want to set, and *value* is an ObjectScript expression that evaluates to the desired value. This could be a constant or could be a more complex expression.

We can use the same syntax to invoke methods of the object (instance methods). An *instance method* is invoked from a specific instance of a class and typically performs some action related to that instance. In the following example, we invoke the **PrintPerson()** method on the object whose *Name* property was just set:

**ObjectScript**

```
set person=##class(Sample.Person).%New()
set person.Name="Carter,Jacob N."
do person.PrintPerson()
```

If the method returns a value, you can use the SET command to assign the returned value to a variable:

```
SET myvar=oref.MethodName()
```

If the method does not return a value (or if you are uninterested in the return value), use either DO or JOB:

```
Do oref.MethodName()
```

If the method accepts arguments, specify them within the parentheses.

**ObjectScript**

```
Set value = oref.methodName(arglist)
```

## 6.5.1 Cascading Dot Syntax

Depending on the class definition, a property can be object-valued, meaning that its type is an object class. In such cases, you can use a chain of OREFs to refer to a property of the properties (or to a method of the property). This is known as *cascading dot syntax*. For example, the following syntax refers to the `Street` property of the `HomeAddress` property of a `Person` object:

```
set person.HomeAddress.Street="15 Mulberry Street"
```

In this example, the `person` variable is an OREF, and the expression `person.HomeAddress` is also an OREF.

**Note:** When referring to a class member generally, sometimes the following informal reference is used: *PackageName.ClassName.Member*, for example, the Accounting.Invoice.LineItem property. This form never appears in code.

### 6.5.2 Cascading Dot Syntax with a Null OREF

When you use a chain of OREFs to refer to a property, and an intermediate object has not been set, it is often desirable to return a null string for the expression instead of receiving an `<INVALID OREF>` error on the intermediate object. Thus if the intermediate object has not been set (is equal to a null string), the value returned for the chained property reference is a null string.

For example, if *pers* is a valid instance of Sample.Person and *pers*.Spouse equals `""`, then the following statement sets the *name* variable to `""`:

```
set name=pers.Spouse.Name
```

If this behavior is not appropriate in some context, then your code must explicitly check the intermediate object reference.

## 6.6 Validating Objects

The %RegisteredObject class provides a way to validate the properties of an instance. An object is *valid* if all of the following are true:

- All required properties have values.

  (To make a property required, you use the Required keyword as described later in this book.

  If a property is of type %Stream, the stream cannot be a null stream. That is, the property is considered to have a value if the **%IsNull()** method returns 0.

- The value of each property, if not null, is valid for the associated property definition.

  For example, if a property is of type %Boolean, the value `"abc"` is not valid, but the values 0 and 1 are.

- The value of each literal property, if not null, obeys all constraints defined by the property definition. The term *constraint* refers to any property keyword that applies a constraint on a property value. For example, *MAXLEN*, *MAXVAL*, *MINVAL*, *VALUELIST*, and *PATTERN* are all constraints; see the chapter "Defining and Using Literal Properties." For example, for a property of type %String, the default value of *MAXLEN*, and this constraints the property to be no more than 50 characters in length.

- (For object-valued properties) All properties of the object obey the preceding rules.

- (For persistent objects) The object does not violate any SQL constraints, such as a uniqueness constraint on a given field.

To determine whether a given object is valid, call its **%ValidateObject()** method. If this method returns 1, then the object is valid. If it returns an error status, the object is not valid. The following shows an example:

```
#include %occStatus
set person=##class(Sample.Person).%New()
set person.DOB="December 12 1990"
set status=person.%ValidateObject()
write !, "First try"
if $$$ISERR(status) {
    do $system.OBJ.DisplayError(status)
} else {
    write !, "Object is valid"
}
```

```
    set person.Name="Ellsworth,Myra Q."
    set person.SSN="000-00-0000"
    set person.DOB=$zdateh("December 12 1990",5)
    set status=person.%ValidateObject()
    write !!, "Second try"
    if $$$ISERR(status) {
        do $system.OBJ.DisplayError(status)
    } else {
        write !, "Object is valid"
    }
```

If you run this example, you will see the following output:

```
First try
ERROR #7207: Datatype value 'December 12 1990' is not a valid number
  > ERROR #5802: Datatype validation failed on property 'Sample.Person:DOB', with value equal to
"December 12 1990"
ERROR #5659: Property 'Sample.Person::Name(1@Sample.Person,ID=)' required
ERROR #5659: Property 'Sample.Person::SSN(1@Sample.Person,ID=)' required
ERROR #7209: Datatype value '' does not match PATTERN '3N1"-"2N1"-"4N'
  > ERROR #5802: Datatype validation failed on property 'Sample.Person:SSN', with value equal to ""

Second try
Object is valid
```

Note that **%ValidateObject()** in turn calls the validation logic for each property; see "Using and Overriding Property Methods" later in this book.

For persistent objects (introduced in the next chapter), when you save an object, the system automatically calls **%ValidateObject()** method first. If the object is not valid, Caché does not save it.

# 6.7 Determining an Object Type

Given an object, the %RegisteredObject class provides methods to determine its inheritance. This section discusses them.

## 6.7.1 %Extends()

To check if an object inherits from a specific superclass, call its **%Extends()** method, and pass the name of that superclass as the argument. If this method returns 1, then the instance inherits from that class. If it returns 0, the instance does not inherit from that class. For example:

```
SAMPLES>set person=##class(Sample.Person).%New()

SAMPLES>w person.%Extends("%RegisteredObject")
1
SAMPLES>w person.%Extends("Sample.Person")
1
SAMPLES>w person.%Extends("Sample.Employee")
0
```

## 6.7.2 %IsA()

To check if an object has a specific class as its *primary* superclass, call its **%IsA()** method, and pass the name of that superclass as the argument. If this method returns 1, the object does have the given class as its primary superclass.

## 6.7.3 %ClassName() and the Most Specific Type Class (MSTC)

Although an object may be an instance of more than one class, it always has a *most specific type class (MSTC)*. A class is said to be the most specific type of an object when that object is an instance of that class and is not an instance of any subclass of that class.

For example, in the case of the GradStudent class inheriting from the Student class that inherits from the Person class, for instances created by the commands:

### ObjectScript

```
set MyInstance1 = ##class(MyPackage.Student).%New()
set MyInstance2 = ##class(MyPackage.GradStudent).%New()
```

*MyInstance1* has Student as its MSTC, since it is an instance of both Person and Student, but not of GradStudent. *MyInstance2* has GradStudent as its MSTC, since it is an instance of GradStudent, Student, and Person.

The following rules also apply regarding the MSTC of an object:

- The MSTC of an object is based solely on primary inheritance.
- A non-instantiable class cannot ever be the MSTC of an object. An object class is non-instantiable if it is abstract.

To determine the MSTC of an object, use the **%ClassName()** method, which is inherited from %RegisteredObject

```
classmethod %ClassName(fullname As %Boolean) as %String
```

Where *fullname* is a boolean argument where 1 specifies that the method return a package name and class name and 0 (the default) specifies that the method return only the class name.

For example:

```
write myinstance.%ClassName(1)
```

(Similarly, you can use **%PackageName()** to get just the name of the package.)

# 6.8 Cloning Objects

To clone an object, call the **%ConstructClone()** method of that object. This method creates a new OREF.

The following Terminal session demonstrates this:

```
SAMPLES>set person=##class(Sample.Person).%OpenId(1)

SAMPLES>set NewPerson=person.%ConstructClone()

SAMPLES>w

NewPerson=<OBJECT REFERENCE>[2@Sample.Person]
person=<OBJECT REFERENCE>[1@Sample.Person]
SAMPLES>
```

Here, you can see that the `NewPerson` variable uses a different OREF than the original `person` object. `NewPerson` is a clone of `person` (or more precisely, these variables are pointers to separate but identical objects).

In contrast, consider the following Terminal session:

```
SAMPLES>set person=##class(Sample.Person).%OpenId(1)

SAMPLES>set NotNew=person

SAMPLES>w

NotNew=<OBJECT REFERENCE>[1@Sample.Person]
person=<OBJECT REFERENCE>[1@Sample.Person]
```

Notice that here, both variables refer to the same OREF. That is, `NotNew` is not a clone of `person`.

For information on arguments to this method, see the InterSystems Class Reference for %Library.RegisteredObject.

# 6.9 Referring to Properties of an Instance

To refer to a property of an instance, you can do any of the following:

- Create an instance of the associated class, and use dot syntax to refer to the property of that instance, as described previously.
- Within an instance method of the associated class, use *relative dot syntax*, as follows:

  ```
  ..PropName
  ```

  You can use this expression with the SET command, or you can use it as part of another expression. The following shows some variations:

  ```
  set value=..PropName
  write ..PropName
  ```

- To access the property, where the property name is not determined until runtime, use the $PROPERTY function. If the property is multidimensional, the following arguments after the property name are used as indices in accessing the value of the property. The signature is:

  ```
  $PROPERTY (oref, propertyName, subscript1, subscript2, subscript3... )
  ```

  Where *oref* is an OREF, *propertyName* evaluates to the name of a property method in the associated class. Also, *subscript1*, *subscript2*, *subscript3* are values for any subscripts of the property; specify these only for a multidimensional property.

  For more information, see the $PROPERTY page in the *Caché ObjectScript Reference*.

- (Within an instance method) use the variable $this, which is introduced later in this chapter.
- (Within an instance method) use instance variables, which are introduced later in this chapter.
- Use the applicable property accessor (getter and setter) methods. See the chapter "Using and Overriding Property Methods."

# 6.10 Calling Methods of an Instance

To call a method of an instance, you can do any of the following:

- Create an instance of the associated class, and use dot syntax to call the method of that instance, as described previously.
- (Within an instance method) to call another instance method of that class (which could be an inherited method), use the following expression:

  ```
  ..MethodName(args)
  ```

  You can use this expression with the DO command. If the method returns a value, you can use SET, or you can use it as part of another expression. The following shows some variations:

  ```
  do ..MethodName()
  set value=..MethodName(args)
  ```

- To execute an instance method, where the method name is not determined until runtime, use the $METHOD function:

  ```
  $METHOD(oref, methodname, Arg1, Arg2, Arg3, ... )
  ```

where *oref* is an OREF, *methodname* evaluates to the name of an instance method in the associated class, and *Arg1*, *Arg2*, *Arg3*, and so on are any arguments to that method.

For more information, see the $METHOD page in the *Caché ObjectScript Reference*.

- (Within an instance method) use the variable $this, which is introduced later in this chapter.

**Important:** If you switch to another namespace, be careful when calling methods of the instance, because the necessary code is not automatically available in the other namespace. For example, suppose that namespaces A and B both have access to the class MyApp.MyObj, but namespace C does not have access to this class. If you create an instance of MyApp.MyObj in namespace A, you can switch to namespace B and then run methods of the instance. If you switch to namespace C, however, you cannot run methods of the instance.

# 6.11 Obtaining the Class Name from an Instance

To obtain the name of a class, use the $CLASSNAME function:

```
$CLASSNAME(oref)
```

where *oref* is an OREF.

For more information, see the $CLASSNAME page in the *Caché ObjectScript Reference*.

# 6.12 $this Variable (Current Instance)

The *$this* syntax provides a handle to the OREF of the current instance, such as for passing it to another class or for another class to refer to properties of methods of the current instance. When an instance refers to its properties or methods, the relative dot syntax is faster and thus is preferred.

**Note:** *$this* is not case-sensitive; hence, *$this*, *$This*, *$THIS*, or any other variant all have the same value.

For example, suppose there is an application with an Accounting.Order class and an Accounting.Utils class. The **Accounting.Order.CalcTax()** method calls the **Accounting.Utils.GetTaxRate()** and **Accounting.Utils.GetTaxableSubtotal()** methods, passing city and state of the current instance to the **GetTaxRate()** method and passing the list of items ordered and relevant tax-related information to **GetTaxableSubtotal()**. **CalcTax()** then uses the values returned to calculate the sales tax for the order. Hence, its code is something like:

### Class Member

```
Method CalcTax() As %Numeric
{
    Set TaxRate = ##Class(Accounting.Utils).GetTaxRate($this)
    Write "The tax rate for ",..City,", ",..State," is ",TaxRate*100,"%",!
    Set TaxableSubtotal = ##class(Accounting.Utils).GetTaxableSubTotal($this)
    Write "The taxable subtotal for this order is $",TaxableSubtotal,!
    Set Tax = TaxableSubtotal * TaxRate
    Write "The tax for this order is $",Tax,!
}
```

The first line of the method uses the *##Class* syntax (described earlier) to invoke the other method; it passes the current object to that method using the *$this* syntax. The second line of the method uses the .. syntax (also described earlier) to get the values of the *City* and *State* properties. The action on the third line is similar to that on the first line.

In the Accounting.Utils, the **GetTaxRate()** method can then use the handle to the passed-in instance to get handles to various properties — for both getting and setting their values:

### Class Member

```
ClassMethod GetTaxRate(OrderBeingProcessed As Accounting.Order) As %Numeric
{
    Set LocalCity = OrderBeingProcessed.City
    Set LocalState = OrderBeingProcessed.State
    // code to determine tax rate based on location and set
    // the value of OrderBeingProcessed.TaxRate accordingly
    Quit OrderBeingProcessed.TaxRate
}
```

The **GetTaxableSubtotal()** method also uses the handle to the instance to look at its properties and set the value of its TaxableSubtotal property.

Hence, if we invoke the **CalcTax()** method of *MyOrder* instance of the Accounting.Order class, we would see something like this:

```
>Do MyOrder.CalcTax()
The tax rate for Cambridge, MA is 5%
The taxable subtotal for this order is $79.82
The tax for this order is $3.99
```

# 6.13 i%PropertyName (Instance Variables)

This section introduces instance variables. You do not need to refer to these variables unless you override an *accessor method* for a property; see the chapter "Using and Overriding Property Methods."

When you create an instance of any class, the system creates an instance variable for each non-calculated property of that class. The instance variable holds the value of the property. For the property *PropName*, the instance variable is named *i%PropName*, and this variable name is case-sensitive. These variables are available within any instance method of the class.

For example, if a class has the properties Name and DOB, then the instance variables *i%Name* and *i%DOB* are available within any instance method of the class.

Internally, Caché also uses additional instance variables with names such as *r%PropName* and *m%PropName*, but these are not supported for direct use.

Instance variables have process-private, in-memory storage allocated for them. Note that these variables are not held in the local variable symbol table and are not affected by the **Kill** command.

# 7

# Introduction to Persistent Objects

This chapter presents the concepts that are useful to understand when working with persistent classes. It discusses the following topics:

- Introduction to persistent classes
- Introduction to the default SQL projection
- Identifiers for saved objects: ID and OID
- Class members specific to persistent classes
- Other class members
- Extents
- Globals

Also see the chapters "Working with Persistent Objects," " Defining Persistent Classes," and "Other Options for Persistent Classes."

When viewing this book online, use the preface of this book to quickly find other topics.

## 7.1 Persistent Classes

A *persistent class* is any class that inherits from %Persistent. A persistent object is an instance of such a class.

The %Persistent class is a subclass of %RegisteredObject and thus is an object class. In addition to providing the methods described in the previous chapter, the %Persistent class defines the *persistence interface*, a set of methods. Among other things, these methods enable you to save objects to the database, load objects from the database, delete objects, and test for existence.

## 7.2 Introduction to the Default SQL Projection

For any persistent class, the compiler generates an SQL table definition, so that the stored data can be accessed via SQL in addition to via the object interface described in this book.

The table contains one record for each saved object, and the table can be queried via Caché SQL. The following shows the results of a query of the Sample.Person table:

| ID | Age | DOB | FavoriteColors | Name | SSN | Spouse | Home_City | Home_State | Home_Street | Home_Zip | Office_City |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 09/19/2013 | Green Green | Humby,Michael G. | 732-63-4007 | | Boston | MD | 4033 Second Avenue | 87184 | Vail |
| 2 | 17 | 08/26/1997 | | Jenkins,Usha V. | 921-79-1526 | | Jackson | KS | 1156 Washington Place | 73402 | Tampa |
| 3 | 16 | 06/02/1998 | White Black | Diavolo,Emma T. | 842-30-8977 | | Larchmont | AL | 5794 Elm Street | 13434 | Miami |
| 4 | 26 | 07/09/1988 | Yellow Purple | Anderson,Alvin T. | 235-56-7318 | | Albany | ND | 4145 Ash Place | 57771 | Vail |
| 5 | 27 | 07/23/1987 | Blue | Zemaitis,Christen D. | 958-46-8312 | | Oak Creek | VT | 6581 Ash Drive | 89034 | Gansevoort |
| 6 | 89 | 08/09/1925 | Red | Adams,Frances Q. | 297-20-9962 | | Chicago | MT | 6141 Washington Blvd | 42799 | Vail |
| 7 | 78 | 04/18/1936 | | Press,Laura F. | 586-94-4188 | | Larchmont | OH | 5772 Washington Drive | 38850 | Islip |

The following table summarizes the default projection:

***Table 7–1: The Object-SQL Projection***

| From (Object Concept) ... | To (Relational Concept) ... |
|---|---|
| Package | Schema |
| Class | Table |
| OID | Identity field |
| Data type property | Field |
| Reference property | Reference field |
| Embedded object | Set of fields |
| List property | List field |
| Array property | Child table |
| Stream property | BLOB |
| Index | Index |
| Class method | Stored procedure |

Later chapters provide more information and describe any changes you can make:

- For information on the table name and the name of the schema to which it belongs, see "Defining Persistent Classes". That chapter also has information on how you can control the projection of subclasses.
- For information on the projection of literal properties, see "Defining and Using Literal Properties."
- For information on the projection of collection properties, see "Working with Collections."
- For information on the projection of stream properties, see "Working with Streams."
- For information on the projection of object-valued properties, see "Defining and Using Object-Valued Properties."
- For information on the projection of relationships, see "Defining and Using Relationships."

# 7.3 Identifiers for Saved Objects: ID and OID

When you save an object for the first time, the system creates two permanent identifiers for it, either of which you can later use to access or remove the saved objects. The more commonly used identifier is the object ID. An ID is a simple literal value that is unique within the table. By default, the system generates an integer to use as an ID.

An OID is more general: it also includes the class name and is unique in the database. In general practice, an application never needs to use the OID value; the ID value is usually sufficient.

The %Persistent class provides methods that use either the ID or the OID. You specify an ID when you use methods such as **%OpenId()**, **%ExistsId()**, and **%DeleteId()**. You specify the OID as the argument for methods such as **%Open()**, **%Exists()**, and **%Delete()**. That is, the methods that use ID as an argument include `Id` in their names. The methods that use OID as the argument do not include `Id` in their names; these methods are used much less often.

When a persistent object is stored in the database, the values of any of its reference attributes (that is, references to other persistent objects) are stored as OID values. For object attributes that do not have OIDs, the literal value of the object is stored along with the rest of the state of the object.

## 7.3.1 Projection of Object IDs to SQL

The ID of an object is available in the corresponding SQL table. If possible, Caché uses the field name `ID`. Caché also provides a way to access the ID if you are not sure what field name to use. The system is as follows:

- An object ID is not a property of the object and is treated differently from the properties.
- If the class does not contain a property named `ID` (in any case variation), then the table also contains the field `ID`, and that field contains the object ID. For an example, see the previous section.
- If the class contains a property that is projected to SQL with the name `ID` (in any case variation), then the table also contains the field `ID1`, and this field holds the value of the object ID.

  Similarly, if the class contains properties that are projected as `ID` and `ID1`, then the table also contains the field `ID2`, and this field holds the value of the object ID.
- In all cases, the table also provides the pseudo-field `%ID`, which holds the value of the object ID.

The OID is not available in the SQL table.

## 7.3.2 Object IDs in SQL

Caché enforces uniqueness for the ID field (whatever its actual name might be). Caché also prevents this field from being changed. This means that you cannot perform SQL **UPDATE** or **INSERT** operations on this field. For instance, the following shows the SQL needed to add a new record to a table:

### SQL

```
INSERT INTO PERSON (FNAME, LNAME)VALUES (:fname, :lname)
```

Notice that this SQL does not refer to the ID field. Caché generates a value for the ID field and inserts that when it creates the requested record.

# 7.4 Class Members Specific to Persistent Classes

Caché classes can include several kinds of class members that are meaningful only in persistent classes. These are storage definitions, indices, foreign keys, and triggers.

## 7.4.1 Storage Definitions

In most cases (as discussed later), each persistent class has a storage definition. The purpose of the storage definition is to describe the global structure that Caché uses when it saves data for the class or reads saved data for the class. Studio displays the storage definition at the end of the class definition, when you view the class in edit mode. The following shows a partial example:

```
<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>
<Value name="5">
<Value>Home</Value>
</Value>
<Value name="6">
<Value>Office</Value>
</Value>
<Value name="7">
<Value>Spouse</Value>
</Value>
<Value name="8">
<Value>FavoriteColors</Value>
</Value>
</Data>
<DataLocation>^Sample.PersonD</DataLocation>
<DefaultData>PersonDefaultData</DefaultData>
<ExtentSize>200</ExtentSize>
<IdLocation>^Sample.PersonD</IdLocation>
<IndexLocation>^Sample.PersonI</IndexLocation>
<Property name="%%CLASSNAME">
<Selectivity>50.0000%</Selectivity>
</Property>
...
```

Also in most cases, the compiler generates and updates the storage definition. For more information on the globals used for persistent classes, see "Globals."

## 7.4.2 Indices

As with other SQL tables, a Caché SQL table can have indices; to define these, you add index definitions to the corresponding class definition.

An index can add a constraint that ensures uniqueness of a given field or combination of fields. For information on such indices, see the chapter "Defining Persistent Classes."

Another purpose of an index is to define a specific sorted subset of commonly requested data associated with a class, so that queries can run more quickly. For example, as a general rule, if a query that includes a WHERE clause using a given field, the query runs more rapidly if that field is indexed. In contrast, if there is no index on that field, the engine must perform a *full table scan*, checking every row to see if it matches the given criteria — an expensive operation if the table is large. See the chapter "Other Options for Persistent Classes."

### 7.4.3 Foreign Keys

A Caché SQL table can also have foreign keys. To define these, you add foreign key definitions to the corresponding class definition.

Foreign keys establish referential integrity constraints between tables that Caché uses when new data is added or when data is changed. If you use relationships, described later in this book, the system automatically treats these as foreign keys. But you can add foreign keys if you do not want to use relationships or if you have other reasons to add them.

For more information on foreign keys, see the chapter "Other Options for Persistent Classes."

### 7.4.4 Triggers

A Caché SQL table can also have triggers. To define these, you add trigger definitions to the corresponding class definition.

Triggers define code to be executed automatically when specific events occur, specifically when a record is inserted, modified, or deleted.

For more information on triggers, see the chapter "Other Options for Persistent Classes."

## 7.5 Other Class Members

A class method or a class query can be defined so that it can be invoked as a stored procedure, which enables you to invoke it from SQL.

For class members not discussed in this chapter, there is no projection to SQL. That is, Caché does not provide a direct way to use them from SQL or to make them usable from SQL.

## 7.6 Extents

The term *extent* refers to all the records, on disk, for a given persistent class. As shown in the next chapter, the %Persistent class provides several methods that operate on the extent of class.

Caché uses an unconventional and powerful interpretation of the object-table mapping. By default, the extent of a given persistent class includes the extents of any subclasses. Therefore:

- If the persistent class Person has the subclass Employee, the Person extent includes all instances of Person and all instances of Employee.
- For any given instance of class Employee, that instance is included in the Person extent and in the Employee 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 Employee instances. Indices defined in the Employee extent contain only Employee 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 Employee extent might include the Department 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 employees, run the same query against the Employee 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.

Similarly, the methods that use the ID all behave polymorphically. That is, they can operate on different types of objects depending on the ID value it is passed.

For example, the extent for Sample.Person objects includes instances of Sample.Person as well as instances of Sample.Employee. When you call **%OpenId()** for the Sample.Person class, the resulting OREF could be an instance of either Sample.Person or Sample.Employee, depending on what is stored within the database:

### ObjectScript

```
// Open person "10"
Set obj = ##class(Sample.Person).%OpenId(10)

Write $ClassName(obj),!    // Sample.Person


// Open person "110"
Set obj = ##class(Sample.Person).%OpenId(110)

Write $ClassName(obj),!    // Sample.Employee
```

Note that the **%OpenId()** method for the Sample.Employee class will *not* return an object if we try to open ID `10`, because the ID 10 is not the Sample.Employee extent:

### ObjectScript

```
// Open employee "10"
Set obj = ##class(Sample.Employee).%OpenId(10)

Write $IsObject(obj),!  // 0

// Open employee "110"
Set obj = ##class(Sample.Employee).%OpenId(110)

Write $IsObject(obj),!  // 1
```

## 7.6.1 Extent Management

For classes that use the default storage class (%Library.CacheStorage), Caché maintains extent definitions and globals that those extents have registered for use with its Extent Manager. The interface to the Extent Manager is through the %ExtentMgr.Util class. This registration process occurs during class compilation. If there are any errors or name conflicts, these cause the compile to fail. For compilation to succeed, resolve the conflicts; this usually involves either changing the name of the index or adding explicit storage locations for the data.

The *MANAGEDEXTENT* class parameter has a default value of 1; this value causes global name registration and a conflicting use check. A value of 0 specifies that there is neither registration nor conflict checking.

**Note:** If an application has multiple classes intentionally sharing a global reference, specify that *MANAGEDEXTENT* equals 0 for all the relevant classes, if they use default storage. Otherwise, recompilation will generate the error such as

```
ERROR #5564: Storage reference: '^This.App.Global used in 'User.ClassA.cls'
is already registered for use by 'User.ClassB.cls'
```

To delete extent metadata, there are multiple approaches:

- Use the `##class(%ExtentMgr.Util).DeleteExtentDefinition(`*`extent`*`,`*`extenttype`*`)` call, where *extent* is typically the class name and *extenttype* is the type of extent (for classes, this is `cls`, which is also the default value for this argument).
- Use one of the following calls:

- $SYSTEM.OBJ.Delete(*classname*,*flags*) where *classname* is the class to delete and *flags* includes e.
- $SYSTEM.OBJ.DeletePackage(*packagename*,*flags*) where *packagename* is the class to delete and *flags* includes e.
- $SYSTEM.OBJ.DeleteAll(*flags*) where *flags* includes e.

These calls are methods of the %SYSTEM.OBJ class.

## 7.6.2 Extent Queries

Every persistent class automatically includes a class query called "Extent" that provides a set of all the IDs in the extent.

For general information on using class queries, see the chapter "Defining and Using Class Queries." The following example uses a class query to display all the IDs for the Sample.Person class:

### ObjectScript

```
set query = ##class(%SQL.Statement).%New()
set status= query.%PrepareClassQuery("Sample.Person","Extent")
if 'status {
  do $system.OBJ.DisplayError(status)
}
set rset=query.%Execute()

While (rset.%Next()) {
    Write rset.%Get("ID"),!
}
```

The Sample.Person extent include all instances of Sample.Person as well as its subclasses. For an explanation of this, see the chapter "Defining Persistent Classes."

The "Extent" query is equivalent to the following SQL query:

```
SELECT %ID FROM Sample.Person
```

Note that you cannot rely on the order in which ID values are returned using either of these methods: Caché may determine that it is more efficient to use an index that is ordered using some other property value to satisfy this request. You can add an ORDER BY %ID clause to the SQL query if you need to.

# 7.7 Globals

Persistent classes allow you to save objects to the database and retrieve them as objects or via SQL. Regardless of how they are accessed, these objects are stored in globals, which can be thought of as persistent multidimensional arrays. For more information on working with globals, see *Using Caché Globals.*

When you define a class that uses the default storage class (%Library.CacheStorage), global names for your class are generated when you compile the class. You can see these names in the storage definition at the bottom of the code in Studio.

The following subsections describe the default global naming scheme, how to generate short global names for better performance, and how to directly control global names.

## 7.7.1 Standard Global Names

When you define a class in Studio, global names for your class are generated based on the class name.

For example, let's define the following class, GlobalsTest.President:

```
Class GlobalsTest.President Extends %Persistent
{

/// President's name (last,first)
Property Name As %String(PATTERN="1U.L1"",""1U.L");

/// Year of birth
Property BirthYear As %Integer;

/// Short biography
Property Bio As %Stream.GlobalCharacter;

/// Index for Name
Index NameIndex On Name;

/// Index for BirthYear
Index DOBIndex On BirthYear;

}
```

After compiling the class, we can see the following storage definition generated at the bottom of the class:

```
Storage Default
{
<Data name="PresidentDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Name</Value>
</Value>
<Value name="3">
<Value>BirthYear</Value>
</Value>
<Value name="4">
<Value>Bio</Value>
</Value>
</Data>
<DataLocation>^GlobalsTest.PresidentD</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<IdLocation>^GlobalsTest.PresidentD</IdLocation>
<IndexLocation>^GlobalsTest.PresidentI</IndexLocation>
<StreamLocation>^GlobalsTest.PresidentS</StreamLocation>
<Type>%Library.CacheStorage</Type>
}
```

Notice, in particular, the following storage keywords:

- The `DataLocation` is the global where class data will be stored. The name of the global is the complete class name (including the package name) with a "D" appended to the name, in this case, `^GlobalsTest.PresidentD`.
- The `IdLocation` (often the same as the `DataLocation`) is the global where the ID counter will be stored, at its root.
- The `IndexLocation` is the global where the indices for the class will be stored. The name of the global is the complete class name with an "I" appended to the name, or, `^GlobalsTest.PresidentI`.
- The `StreamLocation` is the global where any stream properties will be stored. The name of the global is the complete class name with an "S" appended to the name, or, `^GlobalsTest.PresidentS`.

After creating and storing a few objects of our class, we can view the contents of these globals in the Terminal:

```
USER>zwrite ^GlobalsTest.PresidentD
^GlobalsTest.PresidentD=3
^GlobalsTest.PresidentD(1)=$lb("",1732,"1","Washington,George")
^GlobalsTest.PresidentD(2)=$lb("",1735,"2","Adams,John")
^GlobalsTest.PresidentD(3)=$lb("",1743,"3","Jefferson,Thomas")

USER>zwrite ^GlobalsTest.PresidentI
^GlobalsTest.PresidentI("DOBIndex",1732,1)=""
^GlobalsTest.PresidentI("DOBIndex",1735,2)=""
^GlobalsTest.PresidentI("DOBIndex",1743,3)=""
^GlobalsTest.PresidentI("NameIndex"," ADAMS,JOHN",2)=""
```

```
^GlobalsTest.PresidentI("NameIndex"," JEFFERSON,THOMAS",3)=""
^GlobalsTest.PresidentI("NameIndex"," WASHINGTON,GEORGE",1)=""

USER>zwrite ^GlobalsTest.PresidentS
^GlobalsTest.PresidentS=3
^GlobalsTest.PresidentS(1)=1
^GlobalsTest.PresidentS(1,0)=239
^GlobalsTest.PresidentS(1,1)="George Washington was born to a moderately prosperous family of planters
 in colonial ..."
^GlobalsTest.PresidentS(2)=1
^GlobalsTest.PresidentS(2,0)=195
^GlobalsTest.PresidentS(2,1)="John Adams was born in Braintree, Massachusetts, and entered Harvard
College at age 1..."
^GlobalsTest.PresidentS(3)=1
^GlobalsTest.PresidentS(3,0)=202
^GlobalsTest.PresidentS(3,1)="Thomas Jefferson was born in the colony of Virginia and attended the
College of Willi..."
```

The subscript of `^GlobalsTest.PresidentD` is the IDKey. Since we did not define one of our indices as the IDKey, the ID is used as the IDKey. For more information on IDs, see "Controlling How IDs Are Generated."

The first subscript of `^GlobalsTest.PresidentI` is the name of the index.

The first subscript of `^GlobalsTest.PresidentS` is the ID of the bio entry, not the ID of the president.

You can also view these globals in the Management Portal (**System Explorer** > **Globals**).

**Important:** Only the first 31 characters in a global name are significant, so if a complete class name is very long, you might see global names like `^package1.pC347.VeryLongCla4F4AD`. The system generates names such as these to ensure that all of the global names for your class are unique. If you plan to work directly with the globals of a class, make sure to examine the storage definition so that you know the actual name of the global. Alternatively, you can control the global names by using the `DEFAULTGLOBAL` parameter in your class definition. See "User-Defined Global Names."

## 7.7.2 Hashed Global Names

The system will generate shorter global names if you set the `USEEXTENTSET` parameter to the value 1. (The default value for this parameter is 0, meaning use the standard global names.) These shorter global names are created from a hash of the package name and a hash of the class name, followed by a suffix. While the standard names are more readable, the shorter names can contribute to better performance.

When you set `USEEXTENTSET` to 1, each index is also assigned to a separate global, instead of using a single index global with different first subscripts. Again, this is done for increased performance.

To use hashed global names for the GlobalsTest.President class we defined earlier, we would add the following to the class definition:

```
/// Use hashed global names
Parameter USEEXTENTSET = 1;
```

After deleting the storage definition and recompiling the class, we can see the new storage definition with hashed global names:

```
Storage Default
{
...
<DataLocation>^Ebnm.EKUy.1</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<ExtentLocation>^Ebnm.EKUy</ExtentLocation>
<IdLocation>^Ebnm.EKUy.1</IdLocation>
<Index name="DOBIndex">
<Location>^Ebnm.EKUy.2</Location>
</Index>
<Index name="IDKEY">
<Location>^Ebnm.EKUy.1</Location>
</Index>
<Index name="NameIndex">
<Location>^Ebnm.EKUy.3</Location>
</Index>
<IndexLocation>^Ebnm.EKUy.I</IndexLocation>
<StreamLocation>^Ebnm.EKUy.S</StreamLocation>
<Type>%Library.CacheStorage</Type>
}
```

Notice, in particular, the following storage keywords:

- The `ExtentLocation` is the hashed value that will be used to calculate global names for this class, in this case, `^Ebnm.EKUy`.
- The `DataLocation` (equivalent to the `IDKEY` index), where class data will be stored, is now the hashed value with a ".1" appended to the name, in this case, `^Ebnm.EKUy.1`.
- Each index now has its own `Location` and thus its own separate global. The name of the IdKey index global is equivalent to the hashed value with a ".1" appended to the name, in this example, `^Ebnm.EKUy.1`. The globals for the remaining indices have ".2" to ".N" appended to the name. Here, the DOBIndex is stored in global `^Ebnm.EKUy.2` and the NameIndex is stored in `^Ebnm.EKUy.3`.
- The `IndexLocation` is the hashed value with ".I" appended to the name, or `^Ebnm.EKUy.I`, however, this global is often not used.
- The `StreamLocation` is the hashed value with ".S" appended to the name, or `^Ebnm.EKUy.S`.

After creating and storing a few objects, the contents of these globals might look as follows, again using the Terminal:

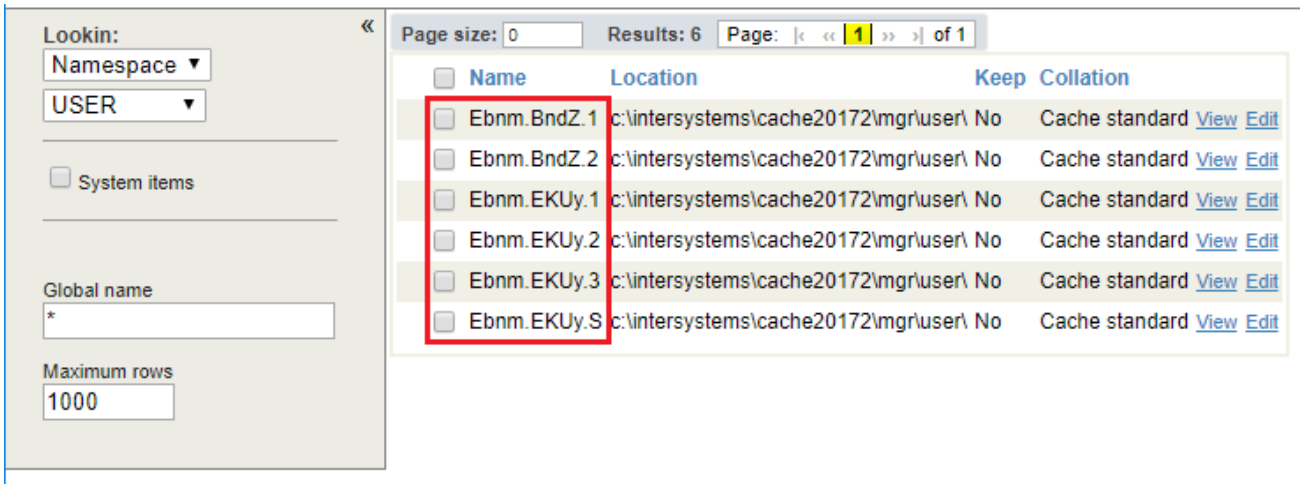
```
USER>zwrite ^Ebnm.EKUy.1
^Ebnm.EKUy.1=3
^Ebnm.EKUy.1(1)=$lb("","Washington,George",1732,"1")
^Ebnm.EKUy.1(2)=$lb("","Adams,John",1735,"2")
^Ebnm.EKUy.1(3)=$lb("","Jefferson,Thomas",1743,"3")

USER>zwrite ^Ebnm.EKUy.2
^Ebnm.EKUy.2(1732,1)=""
^Ebnm.EKUy.2(1735,2)=""
^Ebnm.EKUy.2(1743,3)=""

USER>zwrite ^Ebnm.EKUy.3
^Ebnm.EKUy.3(" ADAMS,JOHN",2)=""
^Ebnm.EKUy.3(" JEFFERSON,THOMAS",3)=""
^Ebnm.EKUy.3(" WASHINGTON,GEORGE",1)=""

USER>zwrite ^Ebnm.EKUy.S
^Ebnm.EKUy.S=3
^Ebnm.EKUy.S(1)=1
^Ebnm.EKUy.S(1,0)=239
^Ebnm.EKUy.S(1,1)="George Washington was born to a moderately prosperous family of planters in colonial
 Virginia..."
...
```

You can also use the `USEEXTENTSET` parameter for classes defined using an SQL **CREATE TABLE** statement. For more information on creating tables, see "Defining Tables" in *Using Caché SQL*.

For example, let's create a table using the Management Portal (**System Explorer** > **SQL** > **Execute Query**):

```
CREATE TABLE GlobalsTest.State (%CLASSPARAMETER USEEXTENTSET 1, NAME CHAR (30) NOT NULL, ADMITYEAR INT)
```

After populating the table with some data, we see the globals ^Ebnm.BndZ.1 and ^Ebnm.BndZ.2 in the Management Portal (**System Explorer** > **Globals**). Notice that the package name is still GlobalsTest, so the first segment of the global names for GlobalsTest.State is the same as for GlobalsTest.President.




Using the Terminal, the contents of the globals might look like:

```
USER>zwrite ^Ebnm.BndZ.1
^Ebnm.BndZ.1=3
^Ebnm.BndZ.1(1)=$lb("Delaware",1787)
^Ebnm.BndZ.1(2)=$lb("Pennsylvania",1787)
^Ebnm.BndZ.1(3)=$lb("New Jersey",1787)

USER>zwrite ^Ebnm.BndZ.2
^Ebnm.BndZ.2(1)=$zwc(412,1,0)/*$bit(2..4)*/
```

The global ^Ebnm.BndZ.1 contains the States data and ^Ebnm.BndZ.2 is a bitmap extent index. See "Bitmap Extent Index" in the *Caché SQL Reference*.

## 7.7.3 User-Defined Global Names

For finer control of the global names for a class, use the DEFAULTGLOBAL parameter. This parameter works in conjunction with the USEEXTENTSET parameter to determine the global naming scheme.

For example, let's add the DEFAULTGLOBAL parameter to set the root of the global names for the GlobalsTest.President class to ^GT.Pres:

```
/// Use hashed global names
Parameter USEEXTENTSET = 1;

/// Set the root of the global names
Parameter DEFAULTGLOBAL = "^GT.Pres";
```

After deleting the storage definition and recompiling the class, we can see the following global names:

```
Storage Default
{
...
<DataLocation>^GT.Pres.1</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<ExtentLocation>^GT.Pres</ExtentLocation>
<IdLocation>^GT.Pres.1</IdLocation>
<Index name="DOBIndex">
<Location>^GT.Pres.2</Location>
</Index>
<Index name="IDKEY">
<Location>^GT.Pres.1</Location>
</Index>
<Index name="NameIndex">
<Location>^GT.Pres.3</Location>
</Index>
<IndexLocation>^GT.Pres.I</IndexLocation>
<StreamLocation>^GT.Pres.S</StreamLocation>
<Type>%Library.CacheStorage</Type>
}
```

Likewise, we can use the `DEFAULTGLOBAL` parameter when defining a class using SQL:

```
CREATE TABLE GlobalsTest.State (%CLASSPARAMETER USEEXTENTSET 0, %CLASSPARAMETER DEFAULTGLOBAL =
'^GT.State',
NAME CHAR (30) NOT NULL, ADMITYEAR INT)
```

This would generate the global names `^GT.StateD` and `^GT.StateI`.

## 7.7.4 Redefining Global Names

If you edit a class definition in a way that redefines the previously existing global names, for example, by changing the values of the `USEEXTENTSET` or `DEFAULTGLOBAL` parameters, you must delete the existing storage definition to allow the compiler to generate a new storage definition. Note that any data in the existing globals is preserved. Any data to be retained must be migrated to the new global structure.

For more information, see “Redefining a Persistent Class That Has Stored Data.”

# 8

# Working with Persistent Objects

The %Persistent class is the API for objects that can be saved (written to disk). This chapter describes how to use this API. Information in this chapter applies to all subclasses of %Persistent. It discusses the following topics:

- How to save objects
- How to test the existence of saved objects
- How to open saved objects
- Swizzling
- How to reloading an object from disk
- How to read stored values
- How to delete saved objects
- How to access object identifiers
- Object concurrency options
- Version checking (alternative to concurrency argument)

Also see the chapters "Introduction to Persistent Objects", "Defining Persistent Classes," and "Other Options for Persistent Classes."

When viewing this book online, use the preface of this book to quickly find other topics.

## 8.1 Saving Objects

To save an object to the database, invoke its **%Save()** method. For example:

### ObjectScript

```
Set obj = ##class(MyApp.MyClass).%New()
Set obj.MyValue = 10

Set sc = obj.%Save()
```

The **%Save()** method returns a %Status value that indicates whether the save operation succeeded or failed. Failure could occur, for example, if an object has invalid property values or violates a uniqueness constraint; see "Validating Objects" in the previous chapter.

Calling **%Save()** on an object automatically saves all modified objects that can be "reached" from the object being saved: that is, all embedded objects, collections, streams, referenced objects, and relationships involving the object are automatically saved if needed. The entire save operation is carried out as one transaction: if any object fails to save, the entire transaction fails and rolls back (no changes are made to disk; all in-memory object values are what they were before calling **%Save()**).

When an object is saved for the first time, the default behavior is for the **%Save()** method to automatically assign it an object ID value that is used to later find the object within the database. In the default case, the ID is generated using the $Increment function; alternately, the class can use a user-provided object ID based on property values that have an idkey index (and, in this case, the property values cannot include the string "||") . Once assigned, you cannot alter the object ID value for a specific object instance (even if it is a user-provided ID).

You can find the object ID value for an object that has been saved using the **%Id()** method:

**ObjectScript**

```
// Open person "22"
Set person = ##class(Sample.Person).%OpenId(22)
Write "Object ID: ",person.%Id(),!
```

In more detail, the **%Save()** method does the following:

1. First it constructs a temporary structure known as a "SaveSet." The SaveSet is simply a graph containing references to every object that is reachable from the object being saved. (Generally, when an object class A has a property whose value is another object class B, an instance of A can "reach" an instance of B.) The purpose of the SaveSet is to make sure that save operations involving complex sets of related objects are handled as efficiently as possible. The SaveSet also resolves any save order dependencies among objects.

   As each object is added to the SaveSet, its **%OnAddToSaveSet()** callback method is called, if present.

2. It then visits each object in the SaveSet in order and checks if they are modified (that is, if any of their property values have been modified since the object was opened or last saved). If an object has been modified, it will then be saved.

3. Before being saved, each modified object is validated (its property values are tested; its **%OnValidateObject()** method, if present, is called; and uniqueness constraints are tested); if the object is valid, the save proceeds. If any object is invalid, then the call to **%Save()** fails and the current transaction is rolled back.

4. Before and after saving each object, the **%OnBeforeSave()** and **%OnAfterSave()** callback methods are called, if present.

   These callbacks are passed an *Insert* argument which indicates whether an object is being inserted (saved for the first time) or updated.

   If either of these callback methods fails (returns an error code) then the call to **%Save()** fails and the current transaction is rolled back.

If the current object is not modified, then **%Save()** does not write it to disk; it returns success because the object did not need to be saved and, therefore, there is no way that there could have been a failure to save it. In fact, the return value of **%Save()** indicates that the save operation either did all that it was asked or it was unable to do as it was asked (and not specifically whether or not anything was written to disk).

**Important:** In a multi-process environment, be sure to use proper concurrency controls; see "Object Concurrency Options."

## 8.1.1 Rollback

The **%Save()** method automatically saves all the objects in its SaveSet as a single transaction. If any of these objects fail to save, then the entire transaction is rolled back. In this rollback, Caché does the following:

1. It reverts assigned IDs.

2. It recovers removed IDs.
3. It recovers modified bits.
4. It invokes the **%OnRollBack()** callback method, if implemented, for any object that had been successfully serialized.

   Caché does not invoke this method for an object that has not been successfully serialized, that is, an object that is not valid.

### 8.1.2 Saving Objects and Transactions

As noted previously, the **%Save()** method automatically saves all the objects in its SaveSet as a single transaction. If any of these objects fail to save, then the entire transaction is rolled back.

If you wish to save two or more *unrelated* objects as a single transaction, however, you must enclose the calls to **%Save()** within an explicit transaction: that is, you must start the transaction using the TSTART command and end it with the TCOMMIT command.

For example:

#### ObjectScript

```
// start a transaction
TSTART

// save first object
Set sc = obj1.%Save()

// save second object (if first was save)
If ($$$ISOK(sc)) {
    Set sc = obj2.%Save()
}

// if both saves are ok, commit the transaction
If ($$$ISOK(sc)) {
    TCOMMIT
}
```

There are two things to note about this example:

1. The **%Save()** method knows if it is being called within an enclosing transaction (because the system variable, $TLEVEL, will be greater than 0).
2. If any of the **%Save()** methods within the transaction fails, the *entire* transaction is rolled back (the TROLLBACK command is invoked). This means that an application must test every call to **%Save()** within a explicit transaction and if one fails, skip calling **%Save()** on the other objects and skip invoking the final **TCOMMIT** command.

## 8.2 Testing the Existence of Saved Objects

There are two basic ways to test if a specific object instance is stored within the database:

- Using ObjectScript
- Using SQL

In these examples, the ID is an integer, which is how Caché generates IDs by default. The next chapter describes how you can define a class so that the ID is instead based on a unique property of the object.

### 8.2.1 Testing for Object Existence with ObjectScript

The **%ExistsId()** class method checks a specified ID; it returns a true value (1) if the specified object is present in the database and false (0) otherwise. It is available to all classes that inherit from %Persistent. For example:

#### ObjectScript

```
Write ##class(Sample.Person).%ExistsId(1),!   // should be 1
Write ##class(Sample.Person).%ExistsId(-1),!  // should be 0
```

Here, the first line should return 1 because Sample.Person inherits from %Persistent and the SAMPLES database provides data for this class.

You can also use the **%Exists()** method, which requires an OID rather than an ID.

### 8.2.2 Testing for Object Existence with SQL

To test for the existence of a saved object with SQL, use a SELECT statement that selects a row whose %ID field matches the given ID. (The identity property of a saved object is projected as the %ID field.)

For example, using embedded SQL:

#### ObjectScript

```
&sql(SELECT %ID FROM Sample.Person WHERE %ID = '1')
Write SQLCODE,!  // should be 0: success

&sql(SELECT %ID FROM Sample.Person WHERE %ID = '-1')
Write SQLCODE,!  // should be 100: not found
```

Here, the first case should result in an SQLCODE of 0 (meaning success) because Sample.Person inherits from %Persistent and the SAMPLES database provides data for this class.

The second case should result in an *SQLCODE* of 100, which means that the statement successfully executed but returned no data. This is expected because the system never automatically generates an ID value less than zero.

For more information on embedded SQL, see the chapter "Embedded SQL" in *Using Caché SQL*. For more information on *SQLCODE*, see "SQLCODE Values and Error Messages" in the *Caché Error Reference*.

## 8.3 Opening Saved Objects

To open an object (load an object instance from disk into memory), use the **%OpenId()** method, which is as follows:

```
classmethod %OpenId(id As %String,
                    concurrency As %Integer = -1,
                    ByRef sc As %Status = $$$OK) as %ObjectHandle
```

Where:

- *id* is the ID of the object to open.

  In these examples, the ID is an integer. The next chapter describes how you can define a class so that the ID is instead based on a unique property of the object.

- *concurrency* is the concurrency level (locking) used to open the object.

- *sc*, which is passed by reference, is a %Status value that indicates whether the call succeeded or failed.

The method returns an OREF if it can open the given object. It returns a null value ("") if it cannot find or otherwise open the object.

For example:

### ObjectScript

```
 // Open person "10"
 Set person = ##class(Sample.Person).%OpenId(10)

 Write "Person: ",person,!    // should be an object reference

 // Open person "-10"
 Set person = ##class(Sample.Person).%OpenId(-10)

 Write "Person: ",person,!    // should be a null string
```

Note that in Caché Basic, the OpenId command is equivalent to the **%OpenId()** method:

```
person = OpenId Sample.Person(1)
PrintLn "Name: " & person.Name
```

You can also use the **%Open()** method, which requires an OID rather than an ID.

## 8.3.1 Multiple Calls to %OpenId()

If **%OpenId()** is called multiple times within a Caché process for the same ID and the same, only one object instance is created in memory: all subsequent calls to **%OpenId()** will return a reference to the object already loaded into memory.

This is demonstrated in the following example:

```
' open and modify Person 1 in memory
personA = OpenId Sample.Person(1)
personA.Name = "Black,Jimmy Carl"

' open Person 1 "again"
personB = OpenId Sample.Person(1)

PrintLn "NameA: " & personA.Name
PrintLn "NameB: " & personB.Name
```

## 8.3.2 Concurrency

The **%OpenId()** method takes an optional *concurrency* argument as input. This argument specifies the concurrency level (type of locks) that should be used to open the object instance.

For more information on the possible object concurrency levels, see "Object Concurrency Options," later in this chapter.

If the **%OpenId()** method is unable to acquire a lock on an object, it will fail.

To raise or lower the current concurrency setting for an object, reopen it with **%OpenId()** and specify a different concurrency level. For example,

### ObjectScript

```
 Set person = ##class(Sample.Person).%OpenId(6,0)
```

opens *person* with a concurrency of 0 and the following effectively upgrades the concurrency to 4:

### ObjectScript

```
 Set person = ##class(Sample.Person).%OpenId(6,4)
```

# 8.4 Swizzling

If you open (load into memory) an instance of a persistent object, and use an object that it references, then this referenced object is automatically opened. This process is referred to as *swizzling*; it is also sometimes known as "lazy loading."

For example, the following code opens an instance of Sample.Employee object and automatically swizzles (opens) its related Sample.Company object by referring to it using dot syntax:

### ObjectScript

```
// Open employee "101"
Set emp = ##class(Sample.Employee).%OpenId(101)

// Automatically open Sample.Company by referring to it:
Write "Company: ",emp.Company.Name,!
```

When an object is swizzled, it is opened using the default concurrency value of the class it is a member of, not the concurrency value of the object that swizzles it. See "Object Concurrency Options," later in this chapter.

A swizzled object is removed from memory as soon as no objects or variables refer to it.

# 8.5 Reloading an Object from Disk

To reload an in-memory object with the values stored within the database, call its **%Reload()** method.

### ObjectScript

```
// Open person "1"
Set person = ##class(Sample.Person).%OpenId(1)
Write "Original value: ",person.Name,!

// modify the object
Set person.Name = "Black,Jimmy Carl"
Write "Modified value: ",person.Name,!

// Now reload the object from disk
Do person.%Reload()
Write "Reloaded value: ",person.Name,!
```

# 8.6 Reading Stored Values

Suppose you have opened an instance of a persistent object, modified its properties, and then wish to view the original value stored in the database before saving the object. The easiest way to do this is to use an SQL statement (SQL is always executed against the database; not against objects in memory).

For example:

### ObjectScript

```
// Open person "1"
Set person = ##class(Sample.Person).%OpenId(1)
Write "Original value: ",person.Name,!

// modify the object
Set person.Name = "Black,Jimmy Carl"
Write "Modified value: ",person.Name,!

// Now see what value is on disk
Set id = person.%Id()
&sql(SELECT Name INTO :name
        FROM Sample.Person WHERE %ID = :id)

Write "Disk value: ",name,!
```

# 8.7 Deleting Saved Objects

The persistence interface includes methods for deleting objects from the database.

## 8.7.1 The %DeleteId() Method

The **%DeleteId()** method deletes an object that is stored within a database, including any stream data associated with the object. This method is as follows:

```
classmethod %DeleteId(id As %String, concurrency As %Integer = -1) as %Status
```

Where:

- *id* is the of the object to open.

  In these examples, the ID is an integer. The next chapter describes how you can define a class so that the ID is instead based on a unique property of the object.

- *concurrency* is the concurrency level (locking) used when deleting the object.

For example:

### ObjectScript

```
Set sc = ##class(MyApp.MyClass).%DeleteId(id)
```

**%DeleteId()** returns a %Status value indicating whether the object was deleted or not.

**%DeleteId()** calls the **%OnDelete()** callback method (if present) before deleting the object. **%OnDelete()** returns a %Status value; if **%OnDelete()** returns an error value, then the object will not be deleted, the current transaction is rolled back, and **%DeleteId()** returns an error value.

Note that the **%DeleteId()** method has no effect on any object instances in memory.

You can also use the **%Delete()** method, which requires an OID rather than an ID.

## 8.7.2 The %DeleteExtent() Method

The **%DeleteExtent()** method deletes every object (and subclass of object) within its extent. Specifically it iterates through the entire extent and invokes the **%DeleteId()** method on each instance.

### 8.7.3 The %KillExtent() Method

The **%KillExtent()** method directly deletes the globals that store an extent of objects (not including data associated with streams). It does not invoke the **%DeleteId()** method and performs no referential integrity actions. This method is simply intended to serve as a help to developers during the development process. (It is similar to the TRUNCATE TABLE command found in older relational database products.) If you need to delete every object in an extent, including associated stream data, use **%DeleteExtent()** instead.

**CAUTION:** **%KillExtent()** is intended for use only in a development environment and should not be used in a live application. **%KillExtent()** bypasses constraints and user-implemented callbacks, potentially causing data integrity problems.

## 8.8 Accessing Object Identifiers

If an object has been saved, it has an ID and an OID, the permanent identifiers that are used on disk. If you have an OREF for the object, you can use that to obtain these identifiers.

To find the ID associated with an OREF, call the **%Id()** method of the object. For example:

**ObjectScript**

```
write oref.%Id()
```

To find the OID associated with an OREF, you have two options:

1. You can call the **%Oid()** method of the object. For example:

   **ObjectScript**

   ```
   write oref.%Oid()
   ```

2. you can access the %%OID property of the object. Because this property name contains % characters, you must enclose the name in double quotes. For example:

   **ObjectScript**

   ```
   write oref."%%OID"
   ```

## 8.9 Object Concurrency Options

It is important to specify concurrency appropriately when you open or delete objects. You can specify concurrency at several different levels:

1. You can specify the *concurrency* argument for the method that you are using.

   Many of the methods of the %Persistent class allow you to specify this argument, an integer. This argument determines how locks are used for concurrency control. A later subsection lists the allowed values.

   If you do not specify the *concurrency* argument, Caché uses the value of the *DEFAULTCONCURRENCY* class parameter for the class you are working with; see the next item.

2. You can specify the *DEFAULTCONCURRENCY* class parameter for the associated class. All persistent classes inherit this parameter from %Persistent, which defines the parameter as an expression that obtains the default concurrency for the process; see the next item.

   You could override this parameter in your class and specify a hardcoded value or an expression that determines the concurrency via your own rules. In either case, the value of the parameter must be one of the allowed concurrency values discussed later in this section.

3. You can set the default concurrency mode for a process. To do so, use the **$system.OBJ.SetConcurrencyMode()** method (which is the **SetConcurrencyMode()** method of the %SYSTEM.OBJ class).

   As in the other cases, you must use one of the allowed concurrency values. If you do not set the concurrency mode for a process explicitly, the default value is 1.

   The **$system.OBJ.SetConcurrencyMode()** method has no effect on any classes that specify an explicit value for the *DEFAULTCONCURRENCY* class parameter.

## 8.9.1 Why Specify Concurrency?

The following scenario demonstrates why it is important to control concurrency appropriately when you read or write objects. Consider the following scenario:

1. Process A opens an object without specifying the concurrency.

```
SAMPLES>set person = ##class(Sample.Person).%OpenId(5)

SAMPLES>write person
1@Sample.Person
```

2. Process B opens the same object with the concurrency value of 4.

```
SAMPLES>set person = ##class(Sample.Person).%OpenId(5, 4)

SAMPLES>write person
1@Sample.Person
```

3. Process A modifies a property of the object and attempts to save it using **%Save()** and receives an error status.

```
SAMPLES>do person.FavoriteColors.Insert("Green")

SAMPLES>set status = person.%Save()

SAMPLES>do $system.Status.DisplayError(status)

ERROR #5803: Failed to acquire exclusive lock on instance of 'Sample.Person'
```

This is an example of concurrent operations without adequate concurrency control. For example, if process A *could* possibly save the object back to the disk, it *must* open the object with concurrency 4 to ensure it can save the object without conflict with other processes. (These values are discussed later in this chapter.) In this case, Process B would then be denied access (failed with a concurrency violation) or would have to wait until Process A releases the object.

## 8.9.2 Concurrency Values

This section describes the possible concurrency values. First, note the following details:

- Atomic writes are guaranteed when concurrency is greater than 0.
- Caché acquires and releases locks during operations such as saving and deleting objects; the details depend upon the concurrency value, what constraints are present, lock escalation status, and the storage structure.
- In all cases, when an object is removed from memory, any locks for it are removed.

The possible concurrency values are as follows; each value has a name, also shown in the list.

### Concurrency Value 0 (No locking)

No locks are used.

### Concurrency Value 1 (Atomic read)

Locks are acquired and released as needed to guarantee that an object read will be executed as an atomic operation.

Caché does not acquire any lock when creating a new object.

While opening an object, Caché acquires a shared lock for the object, *if that is necessary to guarantee an atomic read*. Caché releases the lock after completing the read operation.

The following table lists the locks that are present in each scenario:

| | When object is created | While object is being opened | After object has been opened | After save operation is complete |
|---|---|---|---|---|
| New object | no lock | N/A | N/A | no lock |
| Existing object | N/A | shared lock, if that is necessary to guarantee an atomic read | no lock | no lock |

### Concurrency Value 2 (Shared locks)

The same as 1 (atomic read) except that opening an object *always* acquires a shared lock (even if the lock is not needed to guarantee an atomic read). The following table lists the locks that are present in each scenario:

| | When object is created | While object is being opened | After object has been opened | After save operation is complete |
|---|---|---|---|---|
| New object | no lock | N/A | N/A | no lock |
| Existing object | N/A | shared lock | no lock | no lock |

### Concurrency Value 3 (Shared/retained locks)

Caché does not acquire any lock when creating a new object.

While opening an existing object, Caché acquires a shared lock for the object.

After saving a new object, Caché has a shared lock for the object.

The following table lists the scenarios:

| | When object is created | While object is being opened | After object has been opened | After save operation is complete |
|---|---|---|---|---|
| New object | no lock | N/A | N/A | shared lock |
| Existing object | N/A | shared lock | shared lock | shared lock |

#### Concurrency Value 4 (Exclusive/retained locks)

When an existing object is opened or when a new object is first saved, Caché acquires an exclusive lock.

The following table lists the scenarios:

| | When object is created | While object is being opened | After object has been opened | After save operation is complete |
|---|---|---|---|---|
| New object | no lock | N/A | N/A | exclusive lock |
| Existing object | N/A | exclusive lock | exclusive lock | exclusive lock |

## 8.9.3 Concurrency and Swizzled Objects

An object referenced by a property is swizzled on access using the default concurrency defined by the swizzled object's class. If the default is not defined for the class, the object is swizzled using the default concurrency mode of the process. The swizzled object does not use the concurrency value of the object that swizzles it.

If the object being swizzled is already in memory, then swizzling does not actually open the object — it simply references the existing in-memory object; in that case, the current state of the object is maintained and the concurrency is unchanged.

There are two ways to override this default behavior:

- Upgrade the concurrency on the swizzled object with a call to the **%Open()** method that specifies the new concurrency. For example:

  **ObjectScript**

  ```
  Do person.Spouse.%OpenId(person.Spouse.%Id(),4,.status)
  ```

  where the first argument to **%OpenId()** specifies the ID, the second specifies the new concurrency, and the third (passed by reference) receives the status of the method.

- Set the default concurrency mode for the process before swizzling the object. For example:

  **ObjectScript**

  ```
  Set olddefault = $system.OBJ.SetConcurrencyMode(4)
  ```

  This method takes the new concurrency mode as its argument and returns the previous concurrency mode.

  When you no longer need a different concurrency mode, reset the default concurrency mode as follows:

  **ObjectScript**

  ```
  Do $system.OBJ.SetConcurrencyMode(olddefault)
  ```

# 8.10 Version Checking (Alternative to Concurrency Argument)

Rather than specifying the concurrency argument when you open or delete an object, you can implement *version checking*. To do so, you specify a class parameter called *VERSIONPROPERTY*. All persistent classes have this parameter. When defining a persistent class, the procedure for enabling version checking is:

1. Create a property of type %Integer that holds the updateable version number for each instance of the class.
2. For that property, set the value of the InitialExpression keyword to 0.
3. For the class, set the value of the *VERSIONPROPERTY* class parameter equal to the name of that property. The value of *VERSIONPROPERTY* cannot be changed to a different property by a subclass.

This incorporates version checking into updates to instances of the class.

When version checking is implemented, the property specified by *VERSIONPROPERTY* is automatically incremented each time an instance of the class is updated (either by objects or SQL). Prior to incrementing the property, Caché compares its in-memory value to its stored value. If they are different, then a concurrency conflict is indicated and an error is returned; if they are the same, then the property is incremented and saved.

**Note:** You can use this set of features to implement optimistic concurrency.

To implement a concurrency check in an SQL update statement for a class where *VERSIONPROPERTY* refers to a property called InstanceVersion, the code would be something like:

```
SELECT InstanceVersion,Name,SpecialRelevantField,%ID
    FROM my_table
    WHERE %ID = :myid

    // Application performs operations on the selected row

UPDATE my_table
    SET SpecialRelevantField = :newMoreSpecialValue
    WHERE %ID = :myid AND InstanceVersion = :myversion
```

where *myversion* is the value of the version property selected with the original data.

# 9

# Defining Persistent Classes

A persistent class is a class that defines persistent objects. This chapter describes how to create such classes. It discusses the following topics:

- Basics of defining a persistent class
- How packages are projected to SQL schemas
- How to specify the table name for a persistent class
- Storage definitions and storage classes
- Schema evolution
- How to reset the storage definition
- How to control how IDs are generated
- How to control the SQL projection of subclasses
- Comments on redefining a class that has stored data

Also see the chapters "Introduction to Persistent Objects," "Working with Persistent Objects," and "Other Options for Persistent Classes."

When viewing this book online, use the preface of this book to quickly find other topics.

## 9.1 Defining a Persistent Class

To define a class that defines persistent objects, ensure that the *primary* (first) superclass of your class is either %Persistent or some other persistent class.

For example:

#### Class Definition

```
Class MyApp.MyClass Extends %Persistent
{
}
```

# 9.2 Projection of Packages to Schemas

For persistent classes, the package is represented in SQL as an SQL schema. For instance, if a class is called Team.Player (the Player class in the "Team" package), the corresponding table is "Team.Player" (the Player table in the "Team" schema).

The default package is "User", which is represented in SQL as the "SQLUser" schema. Hence, a class called User.Person corresponds to a table called SQLUser.Person.

If a package name contains periods, the corresponding table name uses an underscore in the place of each. For example, the class MyTest.Test.MyClass (the MyClass class in the "MyTest.Test" package) becomes the table MyTest_Test.MyClass (the MyClass table in the "MyTest_Test" schema).

If an SQL table name is referenced without the schema name, the default schema name (SQLUser) is used. For instance, the command:

**SQL**

```
Select ID, Name from Person
```

is the same as:

**SQL**

```
Select ID, Name from SQLUser.Person
```

# 9.3 Specifying the Table Name for a Persistent Class

For a persistent class, by default, the short class name becomes the table name.

To specify a different table name, use the SqlTableName class keyword. For example:

```
Class App.Products Extends %Persistent [ SqlTableName = NewTableName ]
```

Although Caché places no restrictions on class names, SQL tables cannot have names that are SQL reserved words. Thus if you create a persistent class with a name that is a reserved word, the class compiler will generate an error message. In this case, you must either rename the class or specify a table name for the projection that differs from the class name, using the technique described here.

# 9.4 Storage Definitions and Storage Classes

The %Persistent class provides the high-level interface for storing and retrieving objects in the database. The actual work of storing and loading objects is performed by what is called a *storage class*.

Every persistent object and every serial object uses a storage class to generate the actual methods used to store, load, and delete objects in a database. These internal methods are referred to as the *storage interface*. The storage interface includes methods such as **%LoadData()**, **%SaveData()**, and **%DeleteData()**. Applications never call these methods directly; instead they are called at the appropriate time by the methods of the persistence interface (such as **%OpenId()** and **%Save()**).

The storage class used by a persistent class is specified by a *storage definition*. A storage definition contains a set of keywords and values that define a storage class as well as additional parameters used by the storage interface.

A persistent class may contain more than one storage definition but only one can be active at a time. The active storage definition is specified using the StorageStrategy keyword of the class. By default, a persistent class has a single storage definition called "Default".

For information on the names of the globals that store the data for a class, see "Globals."

### 9.4.1 Updates to a Storage Definition

The storage definition for a class is created when the class is first compiled. Class projection, such as for SQL or MultiValue, occurs after compilation. If a class compiles properly and then projection fails, Caché does not remove the storage definition. Also, if a class is changed in such a way that might affect the storage definition, it is the responsibility of the application developer to determine if the storage definition has been updated and, if necessary, to modify the storage definition to reflect the change. See "Resetting the Storage Definition."

### 9.4.2 The %CacheStorage Storage Class

%CacheStorage is the default storage class used by persistent objects. It automatically creates and maintains a default storage structure for a persistent class.

New persistent classes automatically use the %CacheStorage storage class. The %CacheStorage class lets you control certain aspects of the storage structure used for a class by means of the various keywords in the storage definition.

Refer to the *Class Definition Reference* for details on the various storage keywords.

Also see "Extent Management" in the previous chapter for information on the *MANAGEDEXTENT* class parameter.

### 9.4.3 The %CacheSQLStorage Storage Class

The %CacheSQLStorage class is a special storage class that uses generated SQL SELECT, INSERT, UPDATE, and DELETE statements to provide object persistence.

%CacheSQLStorage is typically used for:

- Mapping objects to preexisting global structures used by older applications.
- Storing objects within an external relational database using the SQL Gateway.

%CacheSQLStorage is more limited than %CacheStorage. Specifically, it does not automatically support schema evolution or multi-class extents.

## 9.5 Schema Evolution

The %CacheStorage storage class supports automatic *schema evolution*.

When you compile a persistent (or serial) class that uses the default %CacheStorage storage class, the class compiler analyzes the properties defined by the class and automatically adds or removes them.

If you would like to see schema evolution in action, try the following:

1. Start Studio and create a new persistent class with one or more properties in it.
2. Compile the class and then view the automatically generated storage definition (as XML text) for the class within the class definition as a whole. Alternatively, you can see a more graphical representation of storage using the Class Inspector. Click on **Storage** in the Inspector, click on "Default" in the list of storage definitions, click on *Data Nodes* in the keyword list, and click on the browse button (...) that appears. This invokes a graphical storage editor.

Within the generated storage for your class, you will see the pseudo-property %%CLASSNAME. This is a placeholder for the class name of any future subclasses you may derive from your class and is used to tell the type of objects stored in the database. For the root class of an extent, this value is always empty.

3. Add one or more new properties to your class and compile it again. Notice that these new properties have been added to your storage definition automatically and in a way that is compatible with the previously existing storage.

# 9.6 Resetting the Storage Definition

During the development process, you may make many modifications to your persistent classes: adding, modifying, and deleting properties. As a result, you may end up with a fairly convoluted storage definition as the class compiler attempts to maintain a compatible structure. If you want the class compiler to generate a clean storage structure, delete the storage definition and recompile the class.

You can do this as follows:

1. Open the class in Studio.
2. Right-click on the default Storage definition in the Class Inspector.
3. Invoke the **Delete** command in the popup menu.
4. Compile the class. This will cause the class compiler to generate a new storage definition for the class.

# 9.7 Controlling How IDs Are Generated

When you save an object for the first time, the system generates an ID for the object. IDs are permanent.

By default, Caché uses an integer for the ID, incremented by 1 from the last saved object.

You can define a given persistent class so that it generates IDs in either of the following ways:

- The ID can be based on a specific property of the class, if that property is unique per instance. For example, you could use a drug code as the ID. To define a class this way, add an index like the following to the class:

  ```
  Index IndexName On PropertyName [ IdKey ];
  ```

  Or (equivalently):

  ```
  Index IndexName On PropertyName [ IdKey, Unique ];
  ```

  Where *IndexName* is the name of the index, and *PropertyName* is the name of the property.

  If you define a class this way, when Caché saves an object for the first time, it uses the value of that property as the ID. Furthermore, Caché requires a value for the property and enforces uniqueness of that property. If you create another object with the same value for the designated property and then attempt to save the new object, Caché issues this error:

  ```
  ERROR #5805: ID key not unique for extent
  ```

  Also, Cache prevents you from changing that property in the future. That is, if you open a saved object, change the property value, and try attempt to save the changed object, Caché issues this error:

  ```
  ERROR #5814: Oid previously assigned
  ```

This message refers to the OID rather than the ID, because the underlying logic prevents the OID from being changed; the OID is based on the ID.

- The ID can be based on multiple properties. To define a class this way, add an index like the following to the class:

  ```
  Index IndexName On (PropertyName1,PropertyName2,...) [ IdKey, Unique ];
  ```

  Or (equivalently):

  ```
  Index IndexName On (PropertyName1,PropertyName2,...) [ IdKey ];
  ```

  Where *IndexName* is the name of the index, and *PropertyName1*, *PropertyName2*, and so on are the property names.

  If you define a class this way, when Caché saves an object for the first time, it generates an ID as follows:

  ```
  PropertyName1||PropertyName2||...
  ```

  Furthermore, Caché requires values for the properties and enforces uniqueness of the given combination of properties. It also prevents you from changing *any* of those properties.

**Important:** If a literal property (that is, an attribute) contains a sequential pair of vertical bars (||), do not add an IdKey index that uses that property. This restriction is imposed by the way in which the Caché SQL mechanism works. The use of || in IdKey properties can result in unpredictable behavior.

The system generates an OID as well. In all cases, the OID has the following form:

```
$LISTBUILD(ID,Classname)
```

Where *ID* is the generated ID, and *Classname* is the name of the class.

# 9.8 Controlling the SQL Projection of Subclasses

When several persistent classes are in superclass/subclass hierarchy, there are two ways in which Caché can store their data. The default scenario is by far the most common.

## 9.8.1 Default SQL Projection of Subclasses

The class compiler projects a "flattened" representation of a persistent class, such that the projected table contains all the appropriate fields for the class, including those that are inherited. Hence, for a subclass, the SQL projection is a table composed of:

- All the columns in the extent of the superclass
- Additional columns based on properties only in the subclass
- Rows that represent the saved instances of the subclass

Furthermore, in the default scenario, the extent of the superclass contains one record for each saved object of the superclass *and all its subclasses*. The extent of each subclass is a subset of the extent of the superclass.

For example, consider the persistent classes Sample.Person and Sample.Employee in SAMPLES. The Sample.Employee class inherits from Sample.Person and adds some additional properties. In the SAMPLES, both classes have saved data.

- The SQL projection of Sample.Person is a table that contains all the suitable properties of the Sample.Person class. The Sample.Person table contains one record for each saved instance of the Sample.Person class *and* each saved instance of the Sample.Employee class.
- The Sample.Employee table includes the same columns as Sample.Person and also includes columns that are specific to the Sample.Employee class. The Sample.Employee table contains one record for each saved instance of the Sample.Employee class.

To see this, use the following SQL queries. The first lists all instances of Sample.Person and shows their properties:

```
SELECT * FROM Sample.Person
```

The second query lists all instances of Sample.Employee and their properties:

```
SELECT * FROM Sample.Employee
```

Notice that the Sample.Person table contains records with IDs in the range 1 to 200. The records with IDs in the range 101 to 200 are employees, and the Sample.Employee table shows the same employees (with the same IDs and with additional columns). The Sample.Person table is arranged in two apparent "groups" *only* because of the artificial way that the SAMPLES database is built. The Sample.Person table is populated and then the Sample.Employee table is populated.

Typically, the table of a subclass has more columns and fewer rows than its parent. There are more columns in the subclass because it usually adds additional properties when it extends the parent class; there are often fewer rows because there are often fewer instances of the subclass than the parent.

## 9.8.2 Alternative SQL Projection of Subclasses

The default projection is the most convenient, but on occasion, you might find it necessary to use the alternative SQL projection. In this scenario, each class has its own extent. To cause this form of projection, include the following in the definition of the superclass:

```
[ NoExtent ]
```

For example:

### Class Definition

```
Class MyApp.MyNoExtentClass [ NoExtent ]
{
//class implementation
}
```

Each subclass of this class then receives its own extent.

If you create classes in this way and use them as properties of other classes, see "Variation: CLASSNAME Parameter" in the chapter "Defining and Using Object-Valued Properties."

# 9.9 Redefining a Persistent Class That Has Stored Data

During the development process, it is common to redefine your classes. If you have already created sample data for the class, note the following points:

- The compiler has no effect on the globals that store the data for the class.

  In fact, when you delete a class definition, its data globals are untouched. If you no longer need these globals, delete them manually.

- If you add or remove properties of a class but do not modify the storage definition of the class, then all code that accesses data for that class continues to work as before. See "Schema Evolution," earlier in this chapter.
- If you do modify the storage definition of the class, then code that accesses the data may or may not continue to work as before, depending on the nature of the change.
- If you modify a property definition in a way that causes the property validation to be more restrictive, then you will receive errors when you work with objects (or records) that no longer pass validation. For example, if you decrease the *MAXLEN* parameter for a property, then you will receive validation errors when you work with an object that has a value for that property that is now too long.

# 10

# Defining and Using Literal Properties

You can define literal properties in any object class. A literal property holds a literal value and is based on a data type class. This chapter describes how to define and use such properties. It discusses the following topics:

- How to define literal properties
- How to define an initial expression for a property
- How to define a property as required (for persistent classes)
- How to define a computed property
- How to define a multidimensional property
- Common data type classes
- Core property parameters
- Class-specific property parameters
- How to define enumerated properties
- How to specify values for literal properties
- How to use property methods
- How to control the SQL projection of literal properties (for persistent classes)

Where noted, some topics also apply to properties of other kinds.

Also see the chapters "Working with Collections," "Working with Streams," "Defining and Using Object-Valued Properties," "Defining and Using Relationships", and "Using and Overriding Property Methods."

When viewing this book online, use the preface of this book to quickly find other topics.

## 10.1 Defining Literal Properties

To add a literal property to a class definition, add an element like one of the following to the class:

### Class Member

```
Property PropName as Classname;

Property PropName as Classname [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) ;
```

Where:

- *PropName* is the name of the property.
- *Classname* is the class on which this property is based. If you omit this, *Classname* is assumed to be %String. To define this property as a literal property, either omit *Classname* or specify *Classname* as the name of a data type class; see "Common Data Type Classes" later in this chapter. You could also use a custom data type class.
- *Keywords* represents any property keywords. See "Compiler Keywords," earlier in this book. Later sections of this chapter discuss additional keywords.
- *PARAM1*, *PARAM2*, and so on are property parameters. The available parameter depend on the class used by the property. Later sections of this chapter lists the most common property parameters.

Notice that the property parameters, if included, are enclosed in parentheses and precede any property keywords. The property keywords, if included, are enclosed in square brackets.

## 10.1.1 Examples

For example, a class can define a Count property using the %Integer data type class:

### Class Member

```
Property Count As %Integer;
```

Because %Integer is a data type class, Count is a data type property.

You can use data type parameters to place constraints on the allowed values of data type properties. For example, for a property of type %Integer, you can specify the *MAXVAL* parameter:

### Class Member

```
Property Count As %Integer(MAXVAL = 100);
```

To set a data type property value equal to the empty string, use code of the form:

### ObjectScript

```
 Set object.Property = $C(0)
```

Every property has a collation type, which determines how values are ordered (such as whether capitalization has effects or not). The default collation type for strings is SQLUPPER. For more details on collations, see "Data Collation" in the chapter "Caché SQL Basics" in *Using Caché SQL*.

# 10.2 Defining an Initial Expression for a Property

By default, when a new object is created, each property equals null. You can specify an ObjectScript expression to use as the initial value for a given property instead. The expression is evaluated when the object is created.

To specify an initial expression for a property, include the InitialExpression keyword in the property definition:

```
Property PropName as Classname [ InitialExpression=expression ] ;
```

Where *expression* is an ObjectScript expression (note that you cannot use any other language for this expression). For details, see InitialExpression in the *Caché Class Definition Reference*.

# 10.3 Defining a Property As Required

This option applies only to properties in persistent classes. (Note that this option applies to any kind of property, not just literal ones.)

By default, when a new object is saved, Caché does not require it to have non-null values for its properties. You can define a property so that a non-null value is required. To do so, include the Required keyword in the property definition:

```
Property PropName as Classname [ Required ] ;
```

Then, if you attempt to save an object that has a null value for the property, Caché issues the following error:

```
ERROR #5659: Property required
```

The Required keyword also affects how you can insert or modify data for this class via SQL access. Specifically, an error occurs if the value is missing when you attempt to insert or update a record.

# 10.4 Defining a Computed Property

In Caché, you can define computed properties, whose values are computed via ObjectScript, possibly based on other properties. The generic phrase *computed properties* (or *computed fields*) includes both of the following variations:

- *Always computed* — The value of this property is calculated when it is accessed. It is never stored in the database.
- *Triggered computed* — The value of this property is recalculated when triggered (details given below).

  If the property is defined in a persistent class, its value *is* stored in the database.

In both cases, the recalculation is performed whether you use object access or an SQL query.

There are five property keywords (SqlComputed, SqlComputeCode, SqlComputeOnChange, Transient, Calculated) that control if and how a property is computed. The following table summarizes the possibilities:

| | | SqlComputed is true (and SqlComputeCode is defined) | SqlComputed is false |
|---|---|---|---|
| Calculated is true | Transient is either true or false | Property is *always computed* | Property is not computed |
| Calculated is false | Transient is true | | |
| | Transient is false | Property is *triggered computed* (SqlComputeOnChange can also be specified in this case) | |

To define a computed property, do the following:

- Include the SqlComputed keyword in the property definition. (That is, specify the SqlComputed keyword as true.)
- Include the SqlComputeCode keyword in the property definition. For the value of this keyword, specify (in curly braces) a line of ObjectScript code that sets the value of the property, according to rules given in "SqlComputeCode" in the reference "Property Keywords" in *Caché Class Definition Reference*. For example:

  ```
  Property FullName As %String [ SqlComputeCode = {set {*}={FirstName}_" "_{LastName}}, SqlComputed
   ];
  ```

- If you want to make the property always computed, specify the Calculated keyword as true in the property definition.

  Or, if you want to make the property triggered computed, do not include the Calculated and Transient keywords in the property definition. (That is, make sure both keywords are false.)

- If the property is triggered computed, optionally specify SqlComputeOnChange.

  This keyword can specify one or more properties. When any of these properties change in value, the triggered property is recomputed. Note that you must use the property names rather than the names given by SqlFieldName, which is discussed later in this chapter). For example (with artificial line breaks):

  ```
  Property messageId As %Integer [
  SqlComputeCode = { set
  {*}=$Select({Status}="":0,1:$List($List($Extract({Status},3,$Length({Status}))))) },
  SqlComputed, SqlComputeOnChange = Status ];
  ```

  For another example (with artificial line breaks):

  ```
  Property Test2 As %String [ SqlComputeCode = { set {*}={Refprop1}_{Refprop2}}, SqlComputed,
  SqlComputeOnChange = (Refprop1, Refprop2) ];
  ```

  The value of SqlComputeOnChange can also include the values `%%INSERT` or `%%UPDATE`; for details, see SqlComputeOnChange.

If you intend to index this field, use deterministic code, rather than nondeterministic code. Caché cannot maintain an index on the results of nondeterministic code because it is not possible to reliably remove stale index key values. (Deterministic code returns the same value every time when passed the same arguments. So for example, code that returns $h is nondeterministic, because $h is modified outside of the control of the function.)

Also see the Calculated keyword in the *Caché Class Definition Reference*. And see "Controlling the SQL Projection of Computed Properties," later in this chapter.

# 10.5 Defining a Multidimensional Property

You can define a property to be multidimensional, which means that you intend the property to act as a multidimensional array. To do so, include the MultiDimensional keyword in the property definition:

```
Property PropName as Classname [ MultiDimensional ] ;
```

This property is different from other properties as follows:

- Caché does not provide property methods for it (see "Using and Overriding Property Methods," later in this book).
- 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 provide a useful way to temporarily contain information about the state of an object.

Also see "Specifying Values for a Multidimensional Property," later in this chapter.

# 10.6 Common Data Type Classes

Caché provides a large number of data type classes. Of these, the classes in most common use are as follows:

***Table 10–1: Common Data Type Classes***

| Class Name | Holds | Notes |
|---|---|---|
| %BigInt | A 64–bit integer | This class is similar to %Integer except for its OdbcType and ClientDataType. |
| %Binary | Binary data | The actual binary data is sent to and from the client without any Unicode (or other) translations. |
| %Boolean | A boolean value | The possible logical values are 0 (false) and 1 (true). |
| %Char | A fixed length character field | |
| %Counter | An integer, meant for use as a unique counter | Any property whose type class is %Counter will be assigned a value when a new object is saved or a new record is inserted via SQL, if no value is specified for that property. For details, see %Counter in the InterSystems Class Reference. |
| %Currency | A currency value | This class is defined only for migrating from Sybase or SQL Server to Caché. |
| %Date | A date | The logical value is in Caché $HOROLOG format. |

| Class Name | Holds | Notes |
|---|---|---|
| %DateTime | A date and time | This class is used mainly for T-SQL migrations and maps datetime/smalldatetime behavior to the %TimeStamp datatype. In this class, the **DisplayToLogical()** and **OdbcToLogical()** methods provide logic to handle imprecise datetime values that are supported by T-SQL applications. |
| %Decimal | A fixed point number | The logical value is a decimal format number. See "Numeric Computing in InterSystems Applications" in the *Caché Programming Orientation Guide*. |
| %Double | An IEEE floating-point number | The logical value is an IEEE floating-point number. See "Numeric Computing in InterSystems Applications" in the *Caché Programming Orientation Guide*. |
| %EnumString | A string | This is a specialized subclass of %String that allows you to define an enumerated set of *possible* values (using the *DISPLAYLIST* and *VALUELIST* parameters). Unlike %String, the display values for this property are used when columns of this type are queried via ODBC. |
| %ExactString | A string | A subclass of %String with the EXACT default collation. |
| %FilemanDate | A date in FileMan format | This class is defined for use with FileMan. See "Converting FileMan Files into Caché Classes" in *Caché Specialized System Tools and Utilities*. |
| %FilemanTimeStamp | A time stamp in FileMan format | This class is defined for use with FileMan. See "Converting FileMan Files into Caché Classes" in *Caché Specialized System Tools and Utilities*. |
| %Float | A floating-point number | This class is deprecated; use %Double or %Decimal instead. |
| %Integer | An integer | |
| %List | Data in $List format | The logical value is data in $List format. |
| %ListOfBinary | Data in $List format, with each list item as binary data | The logical value is data in $List format. |
| %Name | A name in the form "Lastname,Firstname" | The %Name data type has special indexing support when used in conjunction with the %CacheStorage class. For details, see %Name in the InterSystems Class Reference. |
| %Numeric | A fixed-point number | |
| %SmallInt | A small integer value | This class is the same as %Integer except for its OdbcType. |
| %Status | An error status code | Many methods in the Caché Class Library return values of type %Status. For information on working with these values, see %Status in the InterSystems Class Reference. |

| Class Name | Holds | Notes |
|---|---|---|
| %String | A string | |
| %Text | Searchable text | This class supports word-based searching and relevance ranking. For details, see %Text and %Text.Text in the InterSystems Class Reference. |
| %Time | A time value | The logical value is the number of seconds past midnight. |
| %TimeStamp | A value for a time and date | The logical value of the %TimeStamp data type is in YYYY-MM-DD HH:MM:SS.nnnnnnnnn format. Note that if *h* is a date/time value in $H format, then you can use the $ZDATETIME as follows to obtain a valid logical value for a %TimeStamp property: `$ZDATETIME(h,3)` |
| %TinyInt | A very small integer value | This class is the same as %Integer except for its OdbcType and its maximum and minimum values. |
| %MV.Date | A MultiValue date | The logical value is a MultiValue date. To convert a Caché date to a MultiValue date, subtract 46385. |
| %MV.Numeric | A number | This data type class handles the MultiValue masked decimal (MD) conversion codes. |

There are many additional data type classes, intended for specialized uses; most of these types are subclasses of the classes listed here. See the InterSystems Class Reference for details.

Each data type class specifies several keywords that control how that data type is used in Caché SQL and how the data type is projected to client systems:

- SqlCategory — Specifies the SQL category to use for the data type when the Caché SQL engine performs operations upon it.
- OdbcType — Specifies the ODBC type used when the data type is accessed via ODBC.
- ClientDataType — Specifies the Java or ActiveX type used when the data type is accessed via client applications.

Therefore, when you choose a data type class, you should choose a class that has the appropriate client projection, if applicable, for your needs. Use the following subsections to help choose a suitable data type class. If there is no suitable class, you can create your own data type class, as described in the chapter "Defining Data Type Classes."

## 10.6.1 Data Type Classes Grouped by SqlCategory

For a data type class, the SqlCategory class keyword specifies the SQL category that Caché SQL uses when operating on values of properties of that type. Operations controlled by SqlCategory include comparison operations (such as greater than, less than, or equal to); other operations may also use it. The following table shows the SqlCategory values of the data types listed in this chapter.

***Table 10–2: Data Type Classes Grouped by SqlCategory***

| Value | Caché Data Type |
|---|---|
| DATE | %Date |
| DOUBLE | %Double, %Float |
| FMDATE | %FilemanDate |
| FMTIMESTAMP | %FilemanTimeStamp |

| Value | Caché Data Type |
|---|---|
| INTEGER | %BigInt, %Boolean, %Counter, %Integer, %SmallInt, %TinyInt |
| MVDATE | %MV.Date |
| NAME | %Name |
| NUMERIC | %Currency, %Decimal, %Numeric, %MV.Numeric |
| STRING | %Binary, %Char, %EnumString, %ExactString, %List, %ListOfBinary, %Status, %String, %Text |
| TIME | %Time |
| TIMESTAMP | %DateTime, %TimeStamp |

For further information on how literal properties are projected to SQL types, see "Data Types" in the *Caché SQL Reference*.

## 10.6.2 Data Type Classes Grouped by OdbcType

For a data type class, the OdbcType class keyword controls how Caché translates logical data values to and from values used by the Caché SQL ODBC interface. The following table shows the OdbcType values of the data types listed in this chapter.

***Table 10–3: Data Type Classes Grouped by OdbcType***

| Value | Caché Data Type |
|---|---|
| BIGINT | %BigInt |
| BIT | %Boolean |
| DATE | %Date, %FilemanDate, %MV.Date |
| DOUBLE | %Double, %Float |
| INTEGER | %Counter, %Integer |
| NUMERIC | %Currency, %Decimal, %Numeric, %MV.Numeric |
| TIME | %Time |
| TIMESTAMP | %DateTime, %FilemanTimeStamp, %TimeStamp |
| VARBINARY | %Binary |
| VARCHAR | %Char, %EnumString, %ExactString, %List, %ListOfBinary, %Name, %Status, %String, %Text |

## 10.6.3 Data Type Classes Grouped by ClientDataType

For a data type class, the ClientDataType class keyword controls how Caché projects a property (of that type) to Java or Active X. The following table show the ClientDataType values of the data types listed in this chapter.

***Table 10–4: Data Type Classes Grouped by ClientDataType***

| Value | Used for |
|---|---|
| BIGINT | %BigInt |
| BINARY | %Binary (or any property requiring that there is no Unicode conversion of data) |
| CURRENCY | %Currency |

| Value | Used for |
|---|---|
| DATE | %Date |
| DOUBLE | %Double and %Float |
| DECIMAL | %Decimal |
| FDATE | %FilemanDate |
| FTIMESTAMP | %FilemanTimeStamp |
| INTEGER | %Counter, %Integer, %SmallInt, %TinyInt |
| LIST | %List, %ListOfBinary |
| MVDATE | %MV.Date |
| NUMERIC | %Numeric, %MV.Numeric |
| TIME | %Time |
| TIMESTAMP | %DateTime, %TimeStamp |
| VARCHAR | %Char, %EnumString, %ExactString, %Name, %String, %Text |

# 10.7 Core Property Parameters

Depending on the property, you can specify parameters of that property, to affect its behavior. For example, parameters can specify minimum and maximum values, formatting for use in display, collation, delimiters for use in specific scenarios, and so on. You can specify parameters either in the Inspector or by directly typing into the class definition. The following shows an example:

```
Property MyProperty as MyType (MYPARAMETER="some value");
```

Some parameters are available for all properties, of any type. These parameters are as follows:

- *CALCSELECTIVITY* — Controls whether the Tune Table facility calculates the *selectivity* for a property. Usually it is best to leave this parameter as the default (1). For details, see "Tune Table" in the *Caché SQL Optimization Guide*.
- *CAPTION* — Caption to use for this property in client applications.
- *EXTERNALSQLNAME* — Used in linked tables, this parameter specifies the name of the field in the external table to which this property is linked. The Link Table wizard specifies this parameter for each property when it generates a class. The name of the SQL field on the remote database may need to differ from property name on the Caché server for various reason, such as because the remote database field name is a reserved word in Caché. For information on linked tables, see "The Link Table Wizard" in *Using Caché SQL*.

  Note that the property parameter *EXTERNALSQLNAME* has a different purpose than the SQLFieldName compiler keyword, and these items can have different values. SQLFieldName specifies the projected SQL field name in the Caché database, and *EXTERNALSQLNAME* is the field name in the remote database.
- *EXTERNALSQLTYPE* — Used in linked tables, this parameter specifies the SQL type of the field in the external table to which this property is linked. The Link Table wizard specifies this parameter for each property when it generates a class. See *EXTERNALSQLNAME*.
- *JAVATYPE* — The Java data type to which this property is projected.

Most property parameters are defined in data type classes and thus are class-specific; see the next section.

# 10.8 Class-Specific Property Parameters

The previous section lists the parameters that are available for all properties. The other available parameters depend on the class used by the property. The following table lists the parameters supported by the data type classes listed in this chapter. The parameters are grouped into three columns: 1) parameters that are found in many data type classes or that are otherwise commonly encountered, 2) parameters that have meaning *only* in XML and SOAP contexts, and 3) parameters that occur in only a few data type classes and that are rarely encountered.

***Table 10–5: Supported Parameters for System Data Type Classes***

| Data Type Class | Common Parameters | Parameters for XML and SOAP | Less Common Parameters |
|---|---|---|---|
| %BigInt | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %Binary | *MAXLEN*, *MINLEN* | *CANONICALXML*, *MTOM*, *XSDTYPE* | |
| %Boolean | | *XSDTYPE* | |
| %Char | *COLLATION*, *CONTENT*, *DISPLAYLIST*, *ESCAPE*, *MAXLEN*, *MINLEN*, *PATTERN*, *TRUNCATE*, *VALUELIST* | *XMLLISTPARAMETER*, *XSDTYPE* | |
| %Counter | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %Currency* | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | |
| %Date | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %DateTime | *DISPLAYLIST*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XMLDEFAULTVALUE*, *XMLTIMEZONE*, *XSDTYPE* | *DATEFORMAT* |
| %Decimal | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | |
| %Double | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | |
| %EnumString | *COLLATION*, *CONTENT*, *DISPLAYLIST*, *ESCAPE*, *MAXLEN*, *MINLEN*, *PATTERN*, *TRUNCATE*, *VALUELIST* | *XSDTYPE* | |
| %ExactString | *COLLATION*, *CONTENT*, *DISPLAYLIST*, *ESCAPE*, *MAXLEN*, *MINLEN*, *PATTERN*, *TRUNCATE*, *VALUELIST* | *XSDLISTPARAMETER*, *XSDTYPE* | |
| %FilemanDate | | *XSDTYPE* | *STRICTDATA* |
| %FilemanTimeStamp | | *XSDTYPE* | *STRICTDATA* |

| Data Type Class | Common Parameters | Parameters for XML and SOAP | Less Common Parameters |
|---|---|---|---|
| %Float | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | |
| %Integer | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | *STRICT* |
| %List | *ODBCDELIMITER* | *XSDTYPE* | |
| %ListOfBinary | *ODBCDELIMITER* | *XSDTYPE* | |
| %Name | *COLLATION*,*MAXLEN* | *XSDTYPE* | *INDEXSUBSCRIPTS* |
| %Numeric | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | |
| %SmallInt | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %Status | | *XSDTYPE* | |
| %String | *COLLATION*, *CONTENT*, *DISPLAYLIST*, *ESCAPE*, *MAXLEN*, *MINLEN*, *PATTERN*, *TRUNCATE*, *VALUELIST* | *XMLLISTPARAMETER*, *XSDTYPE* | |
| %Text | *COLLATION*, *CONTENT*, *DISPLAYLIST*, *ESCAPE*, *MAXLEN*, *MINLEN*, *PATTERN*, *TRUNCATE*, *VALUELIST* | *XMLLISTPARAMETER*, *XSDTYPE* | *LANGUAGECLASS*, *SIMILARITYINDEX* |
| %Time | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XMLTIMEZONE*, *XSDTYPE* | *PRECISION* |
| %TimeStamp | *DISPLAYLIST*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XMLDEFAULTVALUE*, *XMLTIMEZONE*, *XSDTYPE* | |
| %TinyInt | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %MV.Date | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *VALUELIST* | *XSDTYPE* | |
| %MV.Numeric | *DISPLAYLIST*, *FORMAT*, *MAXVAL*, *MINVAL*, *SCALE*, *VALUELIST* | *XSDTYPE* | *DESCALE* |

*This special-purpose class is only for use in migrations to Caché.

**Note:** The term *constraint* refers to any keyword that applies a constraint on a property value. For example, *MAXVAL*, *MINVAL*, *DISPLAYLIST*, *VALUELIST*, and *PATTERN* are all constraints.

## 10.8.1 Common Parameters

The common parameters from the preceding table are as follows:

- *COLLATION* — Specifies the manner in which property values are transformed for indexing.

The allowable values for collation are discussed in "SQL Introduction" in *Using Caché SQL*.

- *CONTENT* — Specifies the contents of the string, when the string is used in a context where it might be interpreted as XML or HTML. Specify `"STRING"` (the default), `"ESCAPE"`, or `"MIXED"`.

  For details, see *Projecting Objects to XML*.

- *DISPLAYLIST* — Used in conjunction with the *VALUELIST* parameter for enumerated (multiple-choice) properties. For more information, see "Defining Enumerated Properties"

- *ESCAPE* — Specifies the type of escaping to be done, if the string is used in certain contexts. Use either `"XML"` (the default) or `"HTML"`.

  By default, the less than, greater than, and ampersand characters are interpreted as `<` `>` and `&` respectively. For further details on `"XML"`, see *Projecting Objects to XML*.

- *FORMAT* — Specifies the format for the display value. For the value of *FORMAT*, use a format string as specified in the *format* argument of the $FNUMBER function. For properties of type %Numeric or %Decimal, you can also use the option `"AUTO"`, which suppresses any trailing zeroes.

- *MAXLEN* — Specifies the maximum number of characters the string can contain. The default is 50. As with many other parameters, this parameter affects how Caché validates data. Note that it also affects how the field is projected to xDBC clients.

- *MAXVAL* — Specifies the maximum allowed logical value for the data type.

- *MINLEN* — Specifies the minimum number of characters the string can contain.

- *MINVAL* — Specifies the minimum allowed logical value for the data type.

- *ODBCDELIMITER* — Specifies the delimiter character used to construct a %List value when it is projected via ODBC.

- *PATTERN* — Specifies a pattern that the string must match. The value of *PATTERN* must be a valid Caché pattern-matching expression. For an overview of pattern matching, see the "Pattern Matching" section in the "Operators" chapter of *Using Caché ObjectScript*.

- *SCALE* — Specifies the number of digits following the decimal point.

- *TRUNCATE* — Specifies whether to truncate the string to *MAXLEN* characters, where 1 is TRUE and 0 is FALSE. This parameter is used by the **Normalize()** and **IsValid()** methods but is not used by xDBC clients.

- *VALUELIST* — Used for enumerated (multiple-choice) properties. For more information, see "Defining Enumerated Properties"

## 10.8.2 Parameters for XML and SOAP

For information on parameters in the column "Parameters for XML and SOAP," see *Projecting Objects to XML*. Also see *Creating Web Services and Web Clients in Caché*.

## 10.8.3 Less Common Parameters

The less common parameters in the preceding table are as follows:

- *STRICT* (for %Integer) requires that value be an integer. By default, if a property is of type %Integer, and you specify a non-integer numeric value, Caché converts the value to an integer. If *STRICT* is 1 for a property, in such a case, Caché does not convert the value; instead validation fails.

- *DATEFORMAT* (for %DateTime) specifies the order of the date parts when a numeric date format is specified for the display or ODBC input value. Valid parameters are mdy, dmy, ymd, ydm, myd, and dym. The default *DATEFORMAT* is mdy.

- *PRECISION* (for %Time) specifies the number of decimal places to retain. If the value is "" (the default), the system retains the number of decimal places that are provided in the source value. If the value is 0, Caché rounds the provided value to the nearest second.
- *DESCALE* (for %MV.Numeric) specifies the number of decimal place to shift (as with the MultiValue MD conversion).
- *INDEXSUBSCRIPTS* (for %Name) specifies the number of subscripts used by the property in indices, using a comma as a delimiter in the property value; the %CacheStorage class uses this number. A value of 2 specifies that the first comma piece of the property value is stored as the first subscript and the second comma piece of the property value is stored as the second subscript.
- *LANGUAGECLASS* (for %Text) specifies the fully qualified name of the language implementation class. For details, see the class reference for %Text.
- *SIMILARITYINDEX* (for %Text) specifies the name of an index on the current property that has the structure expected by the **SimilarityIdx()** class method of the class specified in the *LANGUAGECLASS* parameter. For details, see the class reference for %Text.
- *STRICTDATA* (for %FilemanDate and %FilemanTimeStamp) affects the generation of the **LogicalToDisplay()** and **LogicalToOdbc()** methods When STRICTDATA=1, imprecise or invalid dates are not changed to a valid FileMan Date value. The default is 0. For example, if Logical FileMan Date value is 31110, by default, this will translate to 3111001 (Sept 01, 2011). If STRICTDATA=1, this transformation does not take place and the invalid/imprecise Logical value gets an error when converted to display or Odbc format.

# 10.9 Defining Enumerated Properties

Many properties support the parameters *VALUELIST* and *DISPLAYLIST*. You use these to define enumerated properties.

To specify a list of valid values for a property, use its *VALUELIST* parameter. The form of *VALUELIST* is a delimiter-separated list of logical values, where the delimiter is the first character. For instance:

### Class Member

```
Property Color As %String(VALUELIST = ",red,green,blue");
```

In this example, *VALUELIST* specifies that valid possible values are “red”, “green”, and “blue”, with a comma as its delimiter. Similarly,

### Class Member

```
Property Color As %String(VALUELIST = " red green blue");
```

specifies the same list, but with a space as its delimiter.

The property is restricted to values in the list, and the data type validation code simply checks to see if the value is in the list. If no list is present, there are no special restrictions on values.

*DISPLAYLIST* is an additional list that, if present, represents the corresponding display values to be returned by the **LogicalToDisplay()** method of the property.

For an example that shows how to obtain the display values, see the section “Using Property Methods,” later in this chapter.

# 10.10 Specifying Values for Literal Properties

To specify a value for a literal property, use the SET command, an OREF, and dot syntax as follows:

```
SET oref.MyProp=value
```

Where *oref* is an OREF, *MyProp* is a property of the corresponding object, and *value* is an ObjectScript expression that evaluates to a literal value. For example:

```
SET patient.LastName="Muggles"
SET patient.HomeAddress.City="Carver"
SET mrn=##class(MyApp.MyClass).GetNewMRN()
set patient.MRN=mrn
```

The literal value must be a valid logical value for the property type. For example, use 1 or 0 for a property based on %Boolean. For another example, for an enumerated property, the value must be one of the items specified by the *VALUELIST* parameter.

## 10.10.1 Specifying Values for a Multidimensional Property

For a multidimensional property, you can specify values for any subscripts of the property. For example:

```
set oref.MyStateProp("temp1")=value1
set oref.MyStateProp("temp2")=value2
set oref.MyStateProp("temp3")=value3
```

Multidimensional properties are useful for holding temporary information for use by the object. These properties are not saved to disk.

# 10.11 Using Property Methods

Each property adds a set of generated class methods to the class. These methods include **propnameIsValid()**, **propnameLogicalToDisplay()**, **propnameDisplayToLogical()**, **propnameLogicalToODBC()**, **propnameODBCToLogical()**, and others, where *propname* is the property name. Some of these methods are inherited from the %Property class and others are inherited from the data type class on which the property is based. For details and a list of the methods, see the chapter "Defining Data Type Classes."

Caché uses these methods internally, and you can call them directly as well. In each case, the argument is a property value. For example:

```
SAMPLES>set p=##class(DeepSee.Study.Patient).%OpenId(1)

SAMPLES>w p.Gender
M
SAMPLES>w ##class(DeepSee.Study.Patient).GenderLogicalToDisplay(p.Gender)
Male
```

# 10.12 Controlling the SQL Projection of Literal Properties

A persistent class is projected as an SQL table, as described earlier in this book. For that class, all properties are projected to SQL, aside from the following exceptions:

- Transient properties (but see the subsection "Controlling the SQL Projection of Computed Properties")
- Calculated properties (but see the subsection "Controlling the SQL Projection of Computed Properties")
- Private properties
- Multidimensional properties

This section discusses the details for literal properties.

## 10.12.1 Specifying the Field Names

By default, a property (if projected to SQL) is projected as an SQL field with the same name as the property. To specify a different field name, use the property keyword SqlFieldName. (Note that it is necessary to use this keyword if the property name is an SQL reserved word). For instance, if there is a %String property called "select," you would define its projected name with the following syntax:

### Class Member

```
Property select As %String [ SqlFieldName = selectfield ];
```

If the name of a property is an SQL reserved word, you need to specify a different name for its projection.

## 10.12.2 Specifying the Column Numbers

The system automatically assigns a unique column number for each field. To control column number assignments, specify the property keyword SqlColumnNumber, as in the following example:

### Class Member

```
Property RGBValue As %String [ SqlColumnNumber = 3 ];
```

The value you specify for SqlColumnNumber must be an integer greater than 1. If you use the SqlColumnNumber keyword without an argument, Caché assigns a column number that is not preserved and that has no permanent position in the table.

If any property has an SQL column number specified, then Caché assigns column numbers for the other properties. The starting value for the assigned column numbers is the number following the highest SQL column number specified.

The value of the SqlColumnNumber keyword is inherited.

## 10.12.3 Effect of the Data Type Class and Property Parameters

The data type class used by a given property has an effect on the SQL projection. Specifically, the SQL category of the data type (defined with the SqlCategory keyword) control how the property is projected. Where applicable, the property parameters also have an effect:

For example, consider the following property definition:

### Class Member

```
Property Name As %String(MAXLEN = 30);}
```

This property is projected as a string field with a maximum length of 30 characters.

## 10.12.4 Controlling the SQL Projection of Computed Properties

In Caché, you can define computed properties, whose values are computed via ObjectScript, possibly based on other properties; for details, see "Defining Computed Properties," earlier in this chapter. The following table summarizes the possibilities and indicates which variations are projected to SQL:

| | | **SqlComputed is true (and SqlComputeCode is defined)** | **SqlComputed is false** |
|---|---|---|---|
| Calculated is true | Transient is true or false | Property is *always computed* and has an SQL projection* | Property is not computed and has no SQL projection |
| Calculated is false | Transient is true | | |
| | Transient is false | Property is *triggered computed* and has an SQL projection* | Property is not computed but does have an SQL projection (this is the default)* |

*This table assumes that the property does not use other keywords that would prevent it from having an SQL projection. For example, it assumes that the property is not private.

# 11

# Working with Collections

Caché supports collections, which provide a way to work with a set of elements, all of the same type. The elements can be literal values or can be objects.

You can define collection properties in any object class. You can also define stand-alone collections for other purposes, such as for use as an method argument or return value. This chapter describes collections, especially collection properties. It discusses the following topics:

- Introduction to collections
- How to define a collection property
- How to specify values for list properties
- How to specify values for array properties
- How to work with list properties
- How to work with array properties
- How to copy collection data
- How to control the SQL projection of collection properties (for persistent classes)
- How to create and use stand-alone collections

Also see the chapters "Defining and Using Literal Properties," "Working with Streams," "Defining and Using Object-Valued Properties," "Defining and Using Relationships", and "Using and Overriding Property Methods."

When viewing this book online, use the preface of this book to quickly find other topics.

## 11.1 Introduction to Collections

A *collection* contains a set of individual elements, all of the same type. There are two kinds of collections: lists and arrays.

Each item in a collection is called an *element* and its position within the collection is called a *key*. For list collections, the system generates sequential integer keys. For arrays, keys can have arbitrary values, and you specify them for each element.

Caché uses a set of collection classes as an interface to collection properties; these are classes in the %Collection package. Caché provides a different set of collection classes for use when you need a stand-alone collection, for example, to pass as an argument to a method; these are classes in the %Library package.

Each set of classes provides methods and properties that you can use to add collection items, remove collection items, count collection items, and so on. This chapter focuses on %Collection classes, but the details are similar for the %Library classes.

Note that collection classes are object classes. Thus a collection is an object.

# 11.2 Defining Collection Properties

To define a list property, add a property as follows:

### Class Member

```
Property MyProp as List of Type;
```

Where *MyProp* is the property name, and *Type* is either a data type class or an object class.

Similarly, to define an array property, add a property as follows:

### Class Member

```
Property MyProp as Array of Type;
```

For example, the following property definition is a list of %String values:

### Class Member

```
Property Colors As List Of %String;
```

For another example, the following property definition is an array of Doctor values, where Doctor is the name of an object class.

### Class Member

```
Property Doctors As Array Of Doctor;
```

*Internally*, Caché uses classes in the %Collection package to represent such properties, as follows:

- %Collection.ListOfDT (if the list element is a data type class)
- %Collection.ListOfObj (if the list element is an object class)
- %Collection.ArrayOfDT (if the array element is a data type class)
- %Collection.ArrayOfObj (if the array element is an object class)

This means that you use methods of these classes to add collection items, remove collection items, and so on. Later parts of this chapter show how this is done.

Do not use the %Collection classes directly as the type of a property. For example, do not create a property definition like this:

```
Property MyProp as %Collection.ArrayOfDT;
```

Instead use the syntax shown earlier in this section.

# 11.3 Adding Items to a List Property

Given a list property (as described in the previous section), use the following procedure to specify a value for the property:

1. If the list items are objects, create those objects as needed.
2. Add list items to the list as needed. To add one list item, call the **Insert()** instance method of the list property. This method is as follows:

   ```
   method Insert(listitem) as %Status
   ```

   Or use other methods of the list property, such as **InsertAt()**. For an introduction, see "Working with List Properties."

   For details on the methods, see the class reference for %Collection.ListOfDT and %Collection.ListOfObj.

For example, suppose that *obj* is an OREF, and *Colors* is a list property of the associated object. In that case, we could add list items as follows:

### ObjectScript

```
Do obj.Colors.Insert("Red")
Do obj.Colors.Insert("Green")
Do obj.Colors.Insert("Blue")
```

For another example, suppose that *pat* is an OREF, and *Diagnoses* is a list property of the associated object. This property is defined as follows, where PatientDiagnosis is the name of a class:

### Class Member

```
Property Diagnoses as list of PatientDiagnosis;
```

In this case, we could add a list item as follows:

### ObjectScript

```
Set patdiag=##class(PatientDiagnosis).%New()
Set patdiag.DiagnosisCode=code
Set patdiag.DiagnosedBy=diagdoc
Set status=pat.Diagnoses.Insert(patdiag)
```

# 11.4 Adding Items to an Array Property

Given an array property (as described earlier in this chapter), use the following procedure to specify a value for the property:

1. If the array items are objects, create those objects as needed.
2. Add array items to the array as needed. To add one list item, call the **SetAt()** instance method of the array property. This method is as follows:

   ```
   method SetAt(element, key As %String) as %Status
   ```

   Where *element* is the element to add, and *key* is the array key to associate with that element.

   **Important:** Do not include a sequential pair of vertical bars (||) within the value that you use as the array key. This restriction is imposed by the way in which the Caché SQL mechanism works.

   For details on this method, see the class reference for %Collection.ArrayOfDT and %Collection.ArrayOfObj. (You will notice that these classes define the same set of methods.)

   Or use the other methods described in "Working with Array Properties."

For example, to add a new color to an array of RGB values accessed by color name in a Palette object, use the following code:

### ObjectScript

```
Do palette.Colors.SetAt("255,0,0","red")
```

where *palette* is the OREF containing the array, *Colors* is the name of the array property, and "red" is the key to access the value "255,0,0".

# 11.5 Working with List Properties

When you create a list property as described earlier, the property itself is an object that provides the instance methods of one of the following classes, depending on the property definition:

- %Collection.ListOfDT (if the list element is a data type class)
- %Collection.ListOfObj (if the list element is an object class)

These classes provide instance methods such as **GetAt()**, **Find()**, **GetPrevious()**, **GetNext()**, and **Remove()**. The following example shows how you might use these methods:

### ObjectScript

```
set p=##class(Sample.Person).%OpenId(1)
for i=1:1:p.FavoriteColors.Count() {
   write !, p.FavoriteColors.GetAt(i)
}
```

Lists are ordered collections of information. Each list element is identified by its position (slot) in the list. You can set the value for a slot or insert data at a slot. If you set a new value for a slot, that value is stored in the list. If you set the value for an already existing slot, the new data overwrites the previous data and the slot assignments are not modified. If you insert data at an already existing slot, the new list item increments the slot number of all subsequent slots. (Inserting a new item in the second slot slides the data currently in the second slot to the third slot, the object currently in the third slot to the fourth slot, and so on.)

You can modify data at slot *n* using the following syntax:

### ObjectScript

```
Do oref.PropertyName.SetAt(data,n)
```

where *oref* is an OREF, *PropertyName* is the name of a list property of that object, and *data* is the actual data. For example, suppose that *person*.FavoriteColors is a list of favorite colors and suppose that this list is initially "red","blue", and "green." To change the second color in the list (so that the list is "red","yellow", and "green"), we can use the following code:

### ObjectScript

```
Do person.FavoriteColors.SetAt("yellow",2)
```

For other methods, such as **Find()**, **RemoveAt()**, and others, see the class reference for %Collection.ListOfDT and %Collection.ListOfObj.

# 11.6 Working with Array Properties

When you create an array property as described earlier, the property itself is an object that provides the instance methods of one of the following classes, depending on the property definition:

- %Collection.ArrayOfDT (if the array element is a data type class)
- %Collection.ArrayOfObj (if the array element is an object class)

These classes provide instance methods such as **GetAt()**, **Find()**, **GetPrevious()**, **GetNext()**, and **Remove()**. For details, see the class reference for these classes. Note that the details are not the same as for the list classes.

# 11.7 Copying Collection Data

To copy the items in one collection into another collection, set the recipient collection equal to the source collection. This copies the contents of the source into the recipient (not the OREF of the collection itself). Some examples of such a command are:

**ObjectScript**

```
Set person2.Colors = person1.Colors
Set dealer7.Inventory = owner3.cars
```

where *person2*, *person1*, *dealer7*, and *owner3* are all instances of classes and *Colors*, *Inventory*, and *cars* are all collection properties. The first line of code looks as it might for copying data between two instances of a single class and the second line of code as it might for copying data from an instance of one class to an instance of a different class.

If the recipient collection is a list and the source collection is an array, Caché copies only the data of the array (not its key values). If the recipient collection is an array and the source collection is a list, the Caché generates key values for the recipient array; these key values are integers based on the position of the item in the source list.

**Note:** There is no way to copy the OREF from one collection to another. It is only possible to copy the data.

# 11.8 Controlling the SQL Projection of Collection Properties

As described earlier in this book, a persistent class is projected as an SQL table. This section describes how list and array properties are projected by default and how you can modify those SQL projections.

## 11.8.1 Default Projection of List Properties

By default, a list property is projected to SQL as a **$LIST** in serialized form. This means that when you obtain such a value, you should use functions suitable for **$LIST** in order to work with it. The following example obtains the value of a list property via embedded SQL and then uses suitable functions to work with the value:

### ObjectScript

```
&sql(SELECT favoritecolors INTO :FavCol FROM Sample.Person WHERE id=1)
write !, $LISTVALID(FavCol)
for i=1:1:$LISTLENGTH(FavCol) {
   write !, $LIST(FavCol,i)
}
```

If the list for a particular instance contains no elements, it is projected as an empty string (and not an SQL NULL value).

## 11.8.2 Default Projection of Array Properties

By default, an array property is projected as a child table, which is in the same package as the parent table. The name of this child table is as follows:

```
tablename_fieldname
```

Where

- *tablename* is the SqlTableName of the parent class (if specified) or the short name of the parent class (if SqlTableName is not specified).
- *fieldname* is the SqlFieldName of the array property (if specified) or the name of the array property (if SqlFieldName is not specified).

For example, a Person class with an array property called Siblings has a projection as a child table called "Person_Siblings".

The child table contains the following three columns:

- One contains the ID of the corresponding instance of the parent class; the name of this column is that of the class containing the array (Person, in the example).
- One contains the identifier for each array member; its name is always element_key.
- One contains array members for all the instances of the class; its name is that of array property (Siblings, in the example).

Continuing the example of the Person class with an array property called Siblings, the projection of Person includes a Person_Siblings child table with the following entries:

***Table 11–1: Sample Projection of an Array Property***

| Person (ID) | element_key | Siblings |
|---|---|---|
| 10 | C | Claudia |
| 10 | T | Tom |
| 12 | B | Bobby |
| 12 | C | Cindy |
| 12 | G | Greg |
| 12 | M | Marsha |
| 12 | P | Peter |

If an instance of the parent class holds an empty collection (one that contains no elements), the ID for that instance does not appear in the child table, such as the instance above where ID equals 11.

Notice that there is no Siblings column in the parent table.

For the column(s) containing the array members, the number and contents of the column(s) depend on the kind of array:

- The projection of an array of data type properties is a single column of data.
- The projection of an array of reference properties is a single column of object references.
- The projection of an array of embedded objects is as multiple columns in the child table. The structure of these columns is described in the section "Embedded Object Properties."

Together, the ID of each instance and the identifier of each array member comprise a unique index for the child table. Also, if a parent instance has no array associated with it, it has no associated entries in the child table.

**Note:** A serial object property is projected to SQL in the same way, by default.

**Important:** When a collection property is projected as an array, there are specific requirements for any index you might add to the property. See "Indexing Collections" in *Caché SQL Optimization Guide*. For an introduction to indices in Caché persistent classes, see the chapter "Other Options for Persistent Classes."

**Important:** There is no support for SQL triggers on child tables projected by array collections. However, if you update the array property and then save the parent object using ObjectScript, any applicable triggers will fire.

## 11.8.3 Alternative Projections of Collections

This section discusses the STORAGEDEFAULT, SQLTABLENAME, and SQLPROJECTION property parameters, which affect how collection properties are stored and projected to SQL.

### 11.8.3.1 STORAGEDEFAULT Parameter

You can store a list property as a child table, and you can store an array property as a $LIST. In both cases, you specify the *STORAGEDEFAULT* parameter of the property:

- For a list property, *STORAGEDEFAULT* is `"list"` by default. If you specify *STORAGEDEFAULT* as `"array"`, then the property is store and projected as a child table. For example:

  ```
  Property MyList as list of %String (STORAGEDEFAULT="array");
  ```

  For details on the resulting projection, see "Default Projection of Array Properties."
- For an array property, *STORAGEDEFAULT* is `"array"` by default. If you specify *STORAGEDEFAULT* as `"list"`, then the property is stored and projected as a $LIST. For example:

  ```
  Property MyArray as array of %String (STORAGEDEFAULT="list");
  ```

  For details on the resulting projection, see "Default Projection of List Properties."

**Important:** The *STORAGEDEFAULT* property parameter affects how the compiler generates storage for the class. If the class definition already includes a storage definition for the given property, the compiler ignores this property parameter.

### 11.8.3.2 SQLTABLENAME Parameter

If a collection property is projected as a child table, you can control the name of that table. To do so, specify the *SQLTABLENAME* parameter of the property. For example:

```
Property MyArray As array Of %String(SQLTABLENAME = "MyArrayTable");
Property MyList As list Of %Integer(SQLTABLENAME = "MyListTable", STORAGEDEFAULT = "array");
```

The *SQLTABLENAME* parameter has no effect unless the property is projected as a child table.

### 11.8.3.3 SQLPROJECTION Parameter

By default, if a collection property is stored as a child table, it is also projected as a child table, but it is not available in the parent table. To make such a property also available in the parent table, specify the *SQLPROJECTION* parameter of the property as `"table/column"`

For example, consider the following class definition:

#### Class Definition

```
Class Sample.Sample Extends %Persistent
{

Property Property1 As %String;

Property Property2 As array Of %String(SQLPROJECTION = "table/column");

}
```

The system generates two tables for this class: Sample.Sample and Sample.Sample_Property2

The table Sample.Sample_Property2 stores the data for the array property Property2, as in the default scenario. Unlike the default scenario, however, a query can refer to the Property2 field in the Sample.Sample table. For example:

```
SAMPLES>>SELECT Property2 FROM Sample.Sample where ID=7
13.     SELECT Property2 FROM Sample.Sample where ID=7

Property2
"1     value 12       value 23       value 3"
```

The `SELECT *` query, however, does not return the Property2 field:

```
SAMPLES>>SELECT * FROM Sample.Sample where ID=7
14.     SELECT * FROM Sample.Sample where ID=7

ID      Property1
7       abc
```

**Note:** There are other possible values of the *SQLPROJECTION* property parameter, but those values have an effect only in MV-enabled classes.

# 11.9 Creating and Using Stand-Alone Collections

The following classes are meant for use as collections that are not class properties:

- %ListOfDataTypes (if the list element is a data type class)
- %ListOfObjects (if the list element is an object class)
- %ArrayOfDataTypes (if the array element is a data type class)
- %ArrayOfObjects (if the array element is an object class)

To create a stand-alone collection, call the **%New()** method of the suitable class to obtain an instance of that class. Then use methods of that instance to add elements and so on. For example:

### ObjectScript

```
set mylist=##class(%ListOfDataTypes).%New()
do mylist.Insert("red")
do mylist.Insert("green")
do mylist.Insert("blue")
write mylist.Count()
```

These classes provide methods with many of the same names as the other collection classes. For details, see the class reference.

# 12

# Working with Streams

Streams provide a way to store large amounts of data (longer than the long string limit). You can define stream properties in any object class. You can also define standalone stream objects for other purposes, such as for use as an method argument or return value. This chapter describes streams and stream properties. It covers the following topics:

- Introduction to stream classes
- How to define stream properties
- How to use the stream interface
- Stream classes for use with gzip files
- How stream properties are projected to SQL (for persistent objects)

When viewing this book online, use the preface of this book to quickly find related topics.

## 12.1 Introduction to Stream Classes

Caché provides the following stream classes:

- %Stream.GlobalCharacter — use this to store character data in global nodes.
- %Stream.GlobalBinary — use this to store binary data in global nodes.
- %Stream.FileCharacter — use this to store character data in an external file.
- %Stream.FileBinary — use to store binary data in an external file.
- %Stream.TmpCharacter — use this when you need a stream to hold character data but do not need to save the data.
- %Stream.TmpBinary — use this when you need a stream to hold binary data but do not need to save the data.

These classes all inherit from %Stream.Object, which defines the common stream interface.

The %Library package also includes stream classes, but those are deprecated. The class library includes additional stream classes, but those are not intended for general use.

Note that stream classes are object classes. Thus a stream is an object.

**Important:** Many of the methods of these classes return status values. In all cases, consult the class reference for details. If a method returns a status value, your code should check that returned value and proceed appropriately. Similarly, for %Stream.FileCharacter and %Stream.FileBinary, if you set the Filename property, you should next check for an error by examining *%objlasterror*.

# 12.2 Declaring Stream Properties

Caché supports both binary streams and character streams. *Binary streams* contain the same sort of data as type %Binary, and are intended for very large binary objects such as pictures. Similarly, *character streams* contain the same sort of data as type %String, and are intended for storing large amounts of text. Character streams, like strings, may undergo a Unicode translation within client applications.

Stream data may be stored in either an external file or a Caché global (or not at all), depending on how the stream property is defined:

- The %Stream.FileCharacter and %Stream.FileBinary classes are used for streams stored as external files.
- The %Stream.GlobalCharacter and %Stream.GlobalBinary classes are used for streams stored as globals.
- The %Stream.TmpCharacter and %Stream.TmpBinary classes are used for streams that do not need to be saved.

The first four classes can use the optional *LOCATION* parameter to specify a default storage location.

In the following example, the JournalEntry class contains four stream properties (one for each of the first four stream classes), and specifies a default storage location for two of them:

### Class Definition

```
Class testPkg.JournalEntry Extends %Persistent
{
Property DailyText As %Stream.FileCharacter;

Property DailyImage As %Stream.FileBinary(LOCATION = "C:/Images");

Property Text As %Stream.GlobalCharacter(LOCATION = "^MyText");

Property Picture As %Stream.GlobalBinary;
}
```

In this example, data for DailyImage is stored in a file (with an automatically generated name) in the C:/Images directory, while the data for the Text property is stored in a global named ^MyText.

# 12.3 Using the Stream Interface

All streams inherit a set of methods and properties used to manipulate the data they contain. The next section lists the commonly used methods and properties, and the following sections provide concrete examples using them:

- Commonly used stream methods and properties
- How to instantiate a stream
- How to read and write stream data
- How to copy between streams
- How to insert stream data
- How to find literal values in a stream
- How to save a stream
- How to use streams in object applications

**Important:** Many of the methods of these classes return status values. In all cases, consult the class reference for details. If a method returns a status value, your code should check that returned value and proceed appropriately. Similarly, for %Stream.FileCharacter and %Stream.FileBinary, if you set the Filename property, you should next check for an error by examining *%objlasterror*.

## 12.3.1 Commonly Used Stream Methods and Properties

Some commonly used methods include the following:

- **Read()** — Read a specified number of characters starting at the current position in the stream.
- **Write()** — Append data to the stream, starting at the current position. Overwrites existing data if the position is not set to the end of the stream.
- **Rewind()** — Move to the beginning of the stream.
- **MoveTo()** — Move to a given position in the stream.
- **MoveToEnd()** — Move to the end of the stream.
- **CopyFrom()** — Copy the contents of a source stream into this stream.
- **NewFileName()** — Specify a filename for a %Stream.FileCharacter or %Stream.FileBinary property.

Commonly used properties include the following:

- AtEnd — Set to `true` when a Read encounters the end of the data source.
- Id — The unique identifier for an instance of a stream within the extent specified by %Location.
- Size — The current size of the stream (in bytes or characters, depending on the type of stream).

For detailed information on individual stream methods and properties, see the InterSystems Class Reference entries for the classes listed at the beginning of this chapter.

## 12.3.2 Instantiating a Stream

When you use a stream class as an object property, the stream is instantiated implicitly when you instantiate the containing object.

When you use a stream class as a standalone object, use the **%New()** method to instantiate the stream.

## 12.3.3 Reading and Writing Stream Data

At the core of the stream interface are the methods **Read()**, **Write()**, and **Rewind()** and the properties AtEnd and Size.

The following example reads data from the Person.Memo stream and writes it to the console, 100 characters at a time. The value of `len` is passed by reference, and is reset to 100 before each Read. The Read method attempts to read the number of characters specified by `len`, and then sets it to the actual number of characters read:

```
Do person.Memo.Rewind()
While (person.Memo.AtEnd = 0) {
  Set len = 100
  Write person.Memo.Read(.len)
}
```

Similarly, you can write data into the stream:

```
Do person.Memo.Write("This is some text. ")
Do person.Memo.Write("This is some more text.")
```

### 12.3.3.1 Specifying a Translation Table

If you are reading or writing a stream of type %Stream.FileCharacter in any character set other than the native character set of the locale, you must set the TranslateTable property of the stream. For an overview of translation tables, see "Default I/O Tables" in the chapter "Localization Support" of the *Caché Programming Orientation Guide*.

## 12.3.4 Copying Data between Streams

All streams contain a **CopyFrom()** method which allows one stream to fill itself from another stream. This can be used, for example, to copy data from a file into a stream property. In this case, one uses the %Library.File class, which is a wrapper around operating system commands and allows you to open a file as a stream. In this case, the code is:

```
// open a text file using a %Library.File stream
Set file = ##class(%File).%New("\data\textfile.txt")
Do file.Open("RU") // same flags as the OPEN command

// Open a Person object containing a Memo stream
// and copy the file into Memo
Set person = ##class(Person).%New()
Do person.Memo.CopyFrom(file)

Do person.%Save() // save the person object
Set person = ""   // close the person object
Set file = ""     // close the file
```

You can also copy data into a stream with the **Set** command:

```
Set person2.Memo = person1.Memo
```

where the Memo property of the Person class holds an OREF for a stream and this command copies the contents of *person1.Memo* into *person2.Memo*.

**Note:** Using **Set** with two streams in this manner does *not* copy the OREF of one stream to the other — it copies the stream contents exclusively. With legacy implementations of streams (through %AbstractStream, %FileBinaryStream, %FileCharacterStream, %GlobalBinaryStream, and %GlobalCharacterStream), the behavior differed: in the previous implementation, the **Set** command in this context copied the OREF.

## 12.3.5 Inserting Stream Data

Apart from the temporary stream classes (whose data cannot be saved), streams have both a temporary and a permanent storage location. All inserts go into the temporary storage area, which is only made permanent when you save the stream. If you start inserting into a stream, then decide that you want to abandon the insert, the data stored in the permanent location is not altered.

If you create a stream, start inserting, then do some reading, you can call **MoveToEnd()** and then continue appending to the temporary stream data. When you save the stream, the data is moved to the permanent storage location.

For example:

```
Set test = ##class(Test).%OpenId(5)
Do test.text.MoveToEnd()
Do test.text.Write("append text")
Do test.%Save()
```

Here, the stream is saved to permanent storage when the *test* object is saved.

## 12.3.6 Finding Literal Values in a Stream

The stream interface includes the **FindAt()** method, which you can use to find the location of a given literal value. This method has the following arguments:

```
method FindAt(position As %Integer, target, ByRef tmpstr, caseinsensitive As %Boolean = 0) as %Integer
```

Where:

- *position* is the position at which to start searching.
- *target* is the literal value to search for.
- *tmpstr*, which is passed by reference, returns information that can be used in the next call to **FindAt()**. Use this when you want to search the same stream repeatedly, starting from the last position where the target was found. In this scenario, specify *position* as –1 and pass *tmpstr* by reference in every call. Then each successive call to **FindAt()** will start where the last call left off.
- *caseinsensitive* specifies whether to perform a case-insensitive search. By default, the method does not consider case.

The method returns the position at this match starting at the beginning of the stream. If it does not find a match, it returns -1.

## 12.3.7 Saving a Stream

When you use a stream class as an object property, the stream data is saved when you save the containing object.

When you use a stream class as a standalone object, use the **%Save()** method to save the stream data. (Note that for the temporary stream classes — %Stream.TmpCharacter and %Stream.TmpBinary — this method returns immediately and does not save any data.)

## 12.3.8 Using Streams in Object Applications

Stream properties are manipulated via a transient object that is created by the object that owns the stream property. Streams act as literal values (think of them as large strings). Two object instances cannot refer to the same stream.

In the following class definition, the Person class has a Memo property that is a stream property:

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

Property Memo As %Stream.GlobalCharacter;
}
```

The following ObjectScript fragment creates a new person object, implicitly instantiating the Memo stream. Then it writes some text to the stream.

```
    // create object and stream
    Set p = ##class(testPkg.Person).%New()
    Set p.Name = "Mo"
    Do p.Memo.Write("This is part one of a long memo. ")
    Do p.Memo.Write("This is part two of a long memo. ")
    Do p.Memo.Write("This is part three of a long memo. ")
    Do p.Memo.Write("This is part four of a long memo. ")
    Do p.%Save()
    Set id = p.%Id() // remember ID for later
    Set p = ""
```

The following code fragment opens the person object and then writes the contents of the stream. Note that when you open an object, the current position of any stream properties is set to the beginning of the stream. This code uses the **Rewind()** method for illustrative purposes.

```
// read object and stream
Set p = ##class(testPkg.Person).%OpenId(id)
Do p.Memo.Rewind() // not required first time

// write contents of stream to console, 100 characters at a time
While (p.Memo.AtEnd = 0) {
    Set len = 100
    Write p.Memo.Read(.len)
}
Set p = ""
```

**Note:** If you want to replace the contents of a stream property, rewind the stream (if the current position of the stream is not already at the beginning), and then use the **Write()** method to write the new data to the stream. Do *not* use the **%New()** method to instantiate a new stream object and assign it to the stream property, for example, `set p.Memo = ##class(%Stream.GlobalCharacter).%New()`, as this leaves the old stream object orphaned in the database.

# 12.4 Stream Classes for Use with gzip Files

The %Stream package also defines the specialized stream classes %Stream.FileBinaryGzip and %Stream.FileCharacterGzip, which you can use to read and write to gzip files. These use the same interface described earlier. Note the following points:

- For these classes, the Size property returns the uncompressed size. When you access the Size property, Caché reads the data in order to calculate the size of the file and this can be an expensive operation
- When you access the Size property, Caché rewinds the stream and leaves you at its start.

# 12.5 Projection of Stream Properties to SQL and ODBC

As described earlier in this book, a persistent class is projected as an SQL table. For such classes, character stream properties and binary stream properties are projected to SQL (and to ODBC clients) as BLOBs (binary large objects).

Stream properties are projected with the ODBC type LONG VARCHAR (or LONG VARBINARY). The ODBC driver/server uses a special protocol to read/write BLOBs. Typically you have to write BLOB applications by hand, because the standard reporting tools do not support them.

The following subsections describes how to use stream properties with SQL. It includes the following topics:

- How to read a stream via embedded SQL
- How to write a stream via embedded SQL

Stream fields have the following restrictions within SQL:

- You cannot use a stream value in a WHERE clause, with a few specific exceptions. For further details, refer to the WHERE clause in the *Caché SQL Reference*.
- You cannot UPDATE/INSERT multiple rows containing a stream; you must do it row by row.

For information on indexing a stream value, contact the InterSystems Worldwide Response Center.

## 12.5.1 Reading a Stream via Embedded SQL

You can use embedded SQL to read a stream as follows:

1. Use embedded SQL to select the OID (Object ID) of the stream:

   ```
   &sql(SELECT Memo INTO :memo FROM Person WHERE Person.ID = 12345)
   ```

   This fetches the OID of the stream and places it into the `memo` host variable.

2. Then open the stream and process it as usual.

## 12.5.2 Writing a Stream via Embedded SQL

To write a stream via embedded SQL, you have several options. For the value to insert, you can use an object reference (OREF) of a stream, the string version of such an OREF, or a string literal.

The following examples show all these techniques. For these examples, assume that you have a table named `Test.ClassWStream` which has a column named `Prop1`, which expects a stream value.

The following example uses an object reference:

### Class Member

```
///use an OREF
ClassMethod Insert1()
{
    set oref=##class(%Stream.GlobalCharacter).%New()
    do oref.Write("Technique 1")

    //do the insert; this time use an actual OREF
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:oref))
}
```

The next example uses a string version of an object reference:

### Class Member

```
///use a string version of an OREF
ClassMethod Insert2()
{
    set oref=##class(%Stream.GlobalCharacter).%New()
    do oref.Write("Technique 2")

    //next line converts OREF to a string OREF
    set string=oref_""

    //do the insert
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:string))
}
```

The last example inserts a string literal into the stream `Prop1`:

### Class Member

```
///insert a string literal into the stream column
ClassMethod Insert3()
{
    set literal="Technique 3"

    //do the insert; use a string
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:literal))
}
```

**Note:** The first character of the string literal cannot be a number. If it is a number, then SQL interprets this as an OREF and attempts to file it as such. Because the stream is not an OREF, this results in an SQL -415 error.

# 13

# Defining and Using Object-Valued Properties

This chapter describes how to define and use object-valued properties, including serial object properties. It discusses the following topics:

- How to define object-valued properties
- Introduction to serial objects
- Possible combinations of objects
- How to specify the value of an object property
- How to save changes
- SQL projection of object-valued properties (for persistent classes)

Relationships provide another way to associate different persistent classes; see the chapter "Relationships." Also see the chapters "Defining and Using Literal Properties," "Working with Collections," "Working with Streams," and "Using and Overriding Property Methods."

When viewing this book online, use the preface of this book to quickly find other topics.

## 13.1 Defining Object-Valued Properties

The phrase *object-valued property* generally refers to a property that is defined as follows:

```
Property PropName as Classname;
```

Where *Classname* is the name of an object class other than a collection or a stream. (Collection properties and stream properties are special cases discussed in earlier chapters.) In general, *Classname* is either a registered object class, a persistent class, or a serial class (see the next section).

To define such a property, define the class to which the property refers and then add the property.

### 13.1.1 Variation: CLASSNAME Parameter

If a property is based on a persistent class, and that class uses the alternative projection of subclasses described in the chapter "Defining Persistent Classes," an additional step is necessary. In this case, it is necessary to specify the *CLASSNAME*

property parameter as 1 for that property. This step affects how Caché stores this property and enables Caché to retrieve the object to which it points.

For example, suppose that MyApp.Payment specifies NoExtent, and MyApp.CreditCard is a subclass of MyApp.Payment. Suppose that MyApp.CurrencyOrder contains a property of type MyApp.CreditCard. That property should specify *CLASSNAME* as 1:

### Class Definition

```
Class MyApp.CurrencyOrder [ NoExtent ]
{
Property Payment as MyApp.CreditCard (CLASSNAME=1);

//other class members
}
```

Note that SQL arrow syntax does not work in this scenario. (You can instead use a suitable JOIN.)

**Important:** Do not specify *CLASSNAME* =1 for a property whose type is a serial class. This usage is not supported.

# 13.2 Introduction to Serial Objects

Serial classes extend %SerialObject. The purpose of such classes is to serve as a property in another object class. The values in a serial object are serialized into the parent object. Serial objects are also called embedded (or embeddable) objects. Caché handles serial object properties differently from non-serial object properties. Two of the differences are as follows:

- It is not necessary to call **%New()** to create the serial object before assigning values to properties in it.
- If the serial object property is contained in a persistent class, the properties of the serial object are stored within the extent of the persistent class.

Later sections of this chapter show these points.

To define a serial class, simply define a class that extends %SerialObject, and add properties and other class members as needed. The following shows an example:

### Class Definition

```
Class Sample.Address Extends %SerialObject
{

/// The street address.
Property Street As %String(MAXLEN = 80);

/// The city name.
Property City As %String(MAXLEN = 80);

/// The 2-letter state abbreviation.
Property State As %String(MAXLEN = 2);

/// The 5-digit U.S. Zone Improvement Plan (ZIP) code.
Property Zip As %String(MAXLEN = 5);

}
```

# 13.3 Possible Combinations of Objects

The following table shows the possible combinations of a parent class and an object-valued property in that class:

| | Property is a registered object class | Property is a persistent class | Property is a serial class |
|---|---|---|---|
| Parent class is a registered object class | Supported | Supported but not common | Supported |
| Parent class is a persistent class | Supported but not common | Supported | Supported |
| Parent class is a serial class | Not supported | Not supported | Supported |

### 13.3.1 Terms for Object-Valued Properties

Within a persistent class, there are two terms for object-valued properties:

- *Reference properties* (properties based on other persistent objects)
- *Embedded object properties* (properties based on serial objects)

Relationships are another kind of property that associates different persistent classes; see the chapter "Relationships." Relationships are bidirectional, unlike the properties described in this chapter.

## 13.4 Specifying the Value of an Object Property

To set an object-valued property, set that property equal to an OREF of an instance of a suitable class.

Consider the scenario where ClassA contains a property PropB that is based on ClassB, where ClassB is an object class:

#### Class Definition

```
Class MyApp.ClassA
{

Property PropB as MyApp.ClassB;

//additional class members
}
```

And ClassB has a non-serial class with its own set of properties Prop1, Prop2, and Prop3.

Suppose that *MyClassAInstance* is an OREF for an instance of ClassA. To set the value of the PropB property for this instance, do the following:

1. If ClassB is not a serial class, first:
   a. Obtain an OREF for an instance of ClassB.
   b. Optionally set properties of this instance. You can also set them later.
   c. Set *MyClassAInstance*.PropB equal to that OREF.

   You can skip this step if ClassB is a serial class.
2. Optionally use cascading dot syntax to set properties of the property (that is, to set properties of *MyClassAInstance*.PropB).

For example:

### ObjectScript

```
set myclassBInstance=##class(MyApp.ClassB).%New()
set myClassBInstance.Prop1="abc"
set myClassBInstance.Prop2="def"
set myClassAInstance.PropB=myclassBInstance
set myClassAInstance.PropB.Prop3="ghi"
```

Notice that this example sets properties of the ClassB instance directly, right after the instance is created, and later more indirectly via cascading dot syntax.

The following steps accomplish the same goal:

### ObjectScript

```
set myClassAInstance.PropB=##class(MyApp.ClassB).%New()
set myClassAInstance.PropB.Prop1="abc"
set myClassAInstance.PropB.Prop2="def"
set myClassAInstance.PropB.Prop3="ghi"
```

In contrast, if ClassB is a serial class, you can do the following, without ever calling **%New()** for ClassB:

### ObjectScript

```
set myClassAInstance.PropB.Prop1="abc"
set myClassAInstance.PropB.Prop2="def"
set myClassAInstance.PropB.Prop3="ghi"
```

# 13.5 Saving Changes

In the case where you are using persistent classes, save the containing object (that is, the instance that contains the object property). There is no need to save the object property directly, because that is saved automatically when the containing object is saved.

The following examples demonstrate these principles. Consider the following persistent classes:

### Class Definition

```
Class MyApp.Customers Extends %Persistent
{

Property Name As %String;

Property HomeStreet As %String(MAXLEN = 80);

Property HomeCity As MyApp.Cities;

}
```

And:

### Class Definition

```
Class MyApp.Cities Extends %Persistent
{

Property City As %String(MAXLEN = 80);

Property State As %String;

Property ZIP As %String;

}
```

In this case, we could create an instance of MyApp.Customers and set its properties as follows:

**ObjectScript**

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="O'Greavy,N."
set customer.HomeStreet="1234 Main Street"
set customer.HomeCity=##class(MyApp.Cities).%New()
set customer.HomeCity.City="Overton"
set customer.HomeCity.State="Any State"
set customer.HomeCity.ZIP="00000"
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

These steps add one new record to MyApp.Customers and one new record to MyApp.Cities.

Instead of calling **%New()** for MyApp.Cities, we could open an existing record:

**ObjectScript**

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="Burton,J.K."
set customer.HomeStreet="17 Milk Street"
set customer.HomeCity=##class(MyApp.Cities).%OpenId(3)
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

In the following variation, we open an existing city and modify it, in the process of adding the new customer:

**ObjectScript**

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="Emerson,S."
set customer.HomeStreet="295 School Lane"
set customer.HomeCity=##class(MyApp.Cities).%OpenId(2)
set customer.HomeCity.ZIP="11111"
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

This change would of course be visible to any other customers with this home city.

# 13.6 SQL Projection of Object-Valued Properties

As described earlier in this book, a persistent class is projected as an SQL table. This section describes how reference properties and embedded object properties of such a class are projected to SQL.

## 13.6.1 Reference Properties

A reference property is projected as a field that contains the ID portion of the OID of the referenced object. For instance, suppose a customer object has a Rep property that refers to a SalesRep object. If a particular customer has a sales representative with an ID of 12, then the entry in the Rep column for that customer is also 12. Because this value matches that of the particular row of the ID column of the referenced object, you can use this value when performing any joins or other processing.

Note that within Caché SQL, you can use a special reference syntax to easily use such references, as an alternative to using a JOIN. For example:

```
SELECT Company->Name FROM Sample.Employee ORDER BY Company->Name
```

## 13.6.2 Embedded Object Properties

An embedded object property is projected as multiple columns in the table of the parent class. One column in the projection contains the entire object in serialized form (including all delimiters and control characters). The rest of the columns are each for one property of the object.

The name of the column for the object property is the same as that of the object property itself. The other column names are made up of the name of the object property, an underscore, and the property within the embedded object. For instance, suppose a class has a Home property containing an embedded object of type Address; Home itself has properties that include Street and Country. The projection of the embedded object then includes the columns named "Home_Street" and "Home_Country". (Note that the column names are derived from the property, Home, and not the type, Address.)

For example, the sample class Sample.Person, includes a Home property which is an embedded object of type Sample.Address. You can use the component fields of Home via SQL as follows:

```
SELECT Name, Home_City, Home_State FROM Sample.Person
WHERE Home_City %STARTSWITH 'B'
ORDER BY Home_City
```

Embedded objects can also include other complex forms of data:

- The projection of a reference property includes a read-only field that includes the object reference as described in "Reference Properties."
- The projection of an array is as a single, non-editable column that is part of the table.
- The projection of a list is as a list field as one of its projected fields; the list field is as described in "Default Projection of List Properties."

# 14

# Defining and Using Relationships

This chapter describes relationships, which are a special kind of property that you can define only in persistent classes. It discusses the following topics:

- Overview
- How to define relationships
- Examples
- How to connect objects in relationships
- How to remove a relationship between objects
- How to delete objects in relationships
- How to work with relationships
- SQL projection of relationships
- How to model many-to-many relationships

When viewing this book online, use the preface of this book to quickly find related topics.

## 14.1 Overview of Relationships

A *relationship* is an association between two persistent objects, each of a specific type. To create a relationship between two objects, each must have a relationship property, which defines its half of the relationship. Caché directly supports two kinds of relationships: one-to-many and parent-child.

Caché relationships have the following characteristics:

- Relationships are *binary* — a relationship is defined either between two, and only two, classes or between a class and itself.
- Relationships can only be defined for *persistent* classes.
- Relationships are *bidirectional* — both sides of a relationship must be defined.
- Relationships automatically provide referential integrity. They are visible to SQL as foreign keys. See "SQL Projection of Relationships" for more information on this topic.
- Relationships automatically manage their in-memory and on-disk behavior.
- Relationships provide superior scaling and concurrency over object collections.

On the other hand, in an object collection, there is an inherent order of the objects; the same is not true for relationships. If you insert objects A, B, and C, in that order, into a list of objects, that order is retained. If you insert objects A, B, and C, in that order, into a relationship property, that order is not retained.

**Note:** It is also possible to define foreign keys between persistent classes, rather than adding relationships. With a foreign key, you have a greater degree of control over what happens when an object in one class is added, updated, or deleted. See "Using Triggers" in *Using Caché SQL*.

## 14.1.1 One-to-Many Relationships

In a one-to-many relationship between class A and class B, one instance of class A is associated with zero or more instances of class B.

For example, a company class may define a one-to-many relationship with an employee class. In this case, there may be zero or more employee objects associated with each company object.

These classes are independent of each other as follows:

- When an instance of either class is created, it may or may not be associated with an instance of the other class.
- If an instance of class B is associated with a given instance of class A, this association can be removed or changed. The instance of class B can be associated with a different instance of class A. The instance of class B does not have to have any association with an instance of class A (and vice versa).

There can be a one-to-many relationship within a single class. One instance of that class can be associated with zero or more other instances of that class. For example, the Employee class might define a relationship between an employee and any employees who directly report that employee.

## 14.1.2 Parent-Child Relationships

In a parent-child relationship between class A and class B, one instance of class A is associated with zero or more instances of class B. Also, the child table is dependent on the parent table, as follows:

- When an instance of the class B is saved, it must be associated with an instance of class A. If you attempt to save the instance, and that association is not defined, the save action fails.
- The association cannot be changed. That is, you cannot associate the instance of class B with a different instance of class A.
- If the instance of class A is deleted, all associated instances of class B are deleted as well.
- You can delete an instance of class B. Class A is not required to have associated instances of class B.

For an example, an invoice class may define a parent-child relationship with a line item class. In this case, an invoice consists of zero or more line items. Those line items cannot be moved to a different invoice. Nor do they have meaning on their own.

**Important:** Also, in the child table (class B), the IDs are not purely numeric. As a consequence, it is not possible to add a bitmap index to the relationship property in this class, although other forms of index are permitted (and are useful, as shown later in this chapter).

#### 14.1.2.1 Parent-Child Relationships and Storage

If you define a parent-child relationship before compiling the classes, the data for both classes is stored in the same global. The data for the children is subordinate to that of the parent, in a structure similar to the following:

```
^Inv(1)
^Inv(1, "invoice", 1)
^Inv(1, "invoice", 2)
^Inv(1, "invoice", 3)
...
```

As a result, Caché can read and write these related objects more quickly.

### 14.1.3 Common Relationship Terminology

This section explains, by example, phrases that are in common use for convenience when discussing relationships.

Consider a one-to-many relationship between a company and its employees; that is, one company has multiple employees. In this scenario, the company is called the *one side* and the employee is called the *many side*.

Similarly, consider a parent-child relationship between a company and its products; that is, the company is the parent, and the products are the children. In this scenario, the company is called the *parent side* and the employee is called the *children side* or the *child side*.

## 14.2 Defining a Relationship

To create a relationship between the records of two classes, you create a pair of complementary relationship properties, one in each class. To create a relationship between records of the same class, you create a pair of complementary relationship properties in that class.

Studio provides a convenient wizard (the New Property Wizard), which simplifies this task. See the "Relationships" section of Using Studio for details.

The following subsections describe the general syntax and then discuss how to define one-to-many relationships and parent-child relationships.

### 14.2.1 General Syntax

The syntax for a relationship property is as follows:

```
Relationship Name As classname [ Cardinality = cardinality_type, Inverse = inverseProp ];
```

Where:

- *classname* is the class to which this relationship refers. This must be a persistent class.
- *cardinality_type* (required) defines how the relationship "appears" from this side as well as whether it is an independent relationship (one-to-many) or a dependent relationship (parent-child). *cardinality_type* can be `one`, `many`, `parent`, or `children`.
- *inverseProp* (required) is the name of the complementary relationship property, which is defined in the other class.

In the complementary relationship property, the *cardinality_type* keyword must be the complement of the *cardinality_type* keyword here. The values `one` and `many` are complements of each other. Similarly, the values `parent` and `children` are complements of each other.

Because a relationship is a kind of property, other property keywords are available for use in them, including Final, Required, SqlFieldName, and Private. Some property keywords, such as MultiDimensional, do not apply. See the *Class Definition Reference* for more information.

## 14.2.2 Defining a One-to-Many Relationship

This section describes how define a one-to-many relationship between *classA* and *classB*, where one instance of *classA* is associated with zero or more instances of *classB*.

**Note:** It is possible to have a one-to-many relationship between records of a single class. That is, in the following discussion, *classA* and *classB* can be the same class.

Class A must have a relationship property of the following form:

### Class Member

```
Relationship manyProp As classB [ Cardinality = many, Inverse = oneProp ];
```

Where *oneProp* is the name of the complementary relationship property, which is defined in *classB*.

Class B must have a relationship property of the following form:

### Class Member

```
Relationship oneProp As classA [ Cardinality = one, Inverse = manyProp ];
```

Where *manyProp* is the name of the complementary relationship property, which is defined in *classA*.

**Important:** On the one side (class A), the relationship uses a query to populate the relationship object. You can improve the performance of this query in almost all cases by adding an index on the complementary relationship property (that is, adding an index on the many side, class B).

The New Property wizard in Studio prompts you to create such an index. See the "Relationships" section of Using Studio for details.

## 14.2.3 Defining a Parent-Child Relationship

This section describes how define a parent-child relationship between *classA* and *classB*, where one instance of *classA* is the parent of zero or more instances of *classB*. These cannot be the same class.

Class A must have a relationship property of the following form:

### Class Member

```
Relationship childProp As classB [ Cardinality = children, Inverse = parentProp ];
```

Where *parentProp* is the name of the complementary relationship property, which is defined in *classB*.

Class B must have a relationship property of the following form:

### Class Member

```
Relationship parentProp As classA [ Cardinality = parent, Inverse = childProp ];
```

Where *childProp* is the name of the complementary relationship property, which is defined in *classA*.

**Important:** On the parent side (class A), the relationship uses a query to populate the relationship object. You can improve the performance of this query in almost all cases by adding an index on the complementary relationship property (that is, adding an index on the child side, class B).

The New Property wizard in Studio prompts you to create such an index. See the "Relationships" section of Using Studio for details.

#### 14.2.3.1 Parent-Child Relationships and Compilation

For a parent-child relationship, Caché can generate a storage definition that stores the data for the parent and child objects within a single global, as shown earlier. Such a storage definition improves the speed with which you can access these related objects.

If you add a relationship after compiling the classes, Caché does not generate this optimized storage definition. In such a case, you can delete any test data you might have, delete the storage definitions of the two classes, and then recompile.

# 14.3 Examples

This section presents examples of a one-to-many relationship and a parent-child relationship.

## 14.3.1 Example One-to-Many Relationship

This example represents a one-to-many relationship between a company and its employees. The company class is as follows:

### Class Definition

```
Class MyApp.Company Extends %Persistent
{

Property Name As %String;

Property Location As %String;

Relationship Employees As MyApp.Employee [ Cardinality = many, Inverse = Employer ];

}
```

And the employee class is as follows:

### Class Definition

```
Class MyApp.Employee Extends (%Persistent, %Populate)
{

Property FirstName As %String;

Property LastName As %String;

Relationship Employer As MyApp.Company [ Cardinality = one, Inverse = Employees ];

Index EmployerIndex On Employer;

}
```

## 14.3.2 Example Parent-Child Relationship

This example represents a parent-child relationship between an invoice and its line items. The invoice class is as follows:

### Class Definition

```
Class MyApp.Invoice Extends %Persistent
{

Property Buyer As %String;

Property InvoiceDate As %TimeStamp;

Relationship LineItems As MyApp.LineItem [ Cardinality = children, Inverse = Invoice ];

}
```

And the line item class is as follows:

### Class Definition

```
Class MyApp.LineItem Extends %Persistent
{

Property ProductSKU As %String;

Property UnitPrice As %Numeric;

Relationship Invoice As MyApp.Invoice [ Cardinality = parent, Inverse = LineItems ];

Index InvoiceIndex On Invoice;

}
```

# 14.4 Connecting Objects

A relationship is bidirectional. Specifically, if you update the value of the relationship property in one object, that immediately affects the value of the corresponding relationship property in the related object. As a consequence, you can specify the value for a relationship property in one object, and the effect appears in both objects.

Because the nature of the relationship property is different in the two classes, there are two general scenarios for updating any relationship:

- Scenario 1: The relationship property is a simple reference property. Set the property equal to the appropriate object.
- Scenario 2: The relationship property is an instance of %RelationshipObject, which has an array-like interface. Use methods of that interface to insert objects into the relationship. Note that the objects in the relationship are *not* ordered; the relationship does not retain the order in which you inserted objects into it.

The following subsections give the details. The third subsection describes a variation of Scenario 1 that is especially suitable when you have a large number of objects in the relationship.

The information here describes how to add objects to relationships. The process of modifying objects is similar, with an important exception (by design) in the case of parent-child relationships: Once associated with a particular parent object (and then saved), a child object can never be associated with a different parent.

## 14.4.1 Scenario 1: Updating the Many or Child Side

On the many side or the child side (*ObjA*), the relationship property is a simple reference property that points to *ObjB*. To connect the objects from this side:

1. Obtain an OREF (*ObjB*) for an instance of the other class. (Either create a new object or open an existing object, as appropriate.)
2. Set the relationship property of *ObjA* equal to *ObjB*.

For an example, consider the example parent-child classes shown earlier. The following steps would update the relationship from the MyApp.LineItem side:

**ObjectScript**

```
 //obtain an OREF to the invoice class
set invoice=##class(MyApp.Invoice).%New()
//...specify invoice date and so on

set item=##class(MyApp.LineItem).%New()
//...set some properties of this object such as the product name and sale price...

//connect the objects
set item.Invoice=invoice
```

When you call the **%Save()** method for the *item* object, the system saves both objects (*item* and *invoice*).

Also see the last subsection for a variation of this technique.

## 14.4.2 Scenario 2: Updating the One or Parent Side

On the one side or the parent side, the relationship property is an instance of %RelationshipObject. On this side, you can do the following to connect the objects:

1. Obtain an OREF for an instance of the other object. (Either create a new object or open an existing object, as appropriate.)
2. Call the **Insert()** method of the relationship property on this side and pass that OREF as the argument.

Consider the example parent-child classes shown earlier. For those classes, the following steps would update the relationship from the MyApp.Invoice side:

**ObjectScript**

```
set invoice=##class(MyApp.Invoice).%OpenId(100034)
//set some properties such as the customer name and invoice date

set item=##class(MyApp.LineItem).%New()
//...set some properties of this object such as the product name and sale price...

//connect the objects
do invoice.LineItems.Insert(item)
```

When you call the **%Save()** method for the *invoice* object, the system saves both objects (*item* and *invoice*).

**Important:** Caché does not maintain information about the order in which objects are added into the relationship. That is, if you open a previously saved object and use **GetNext()** or similar methods to iterate through a relationship, the order of objects in that relationship is different from when the objects were created.

## 14.4.3 Fastest Way to Connect Objects

When you need to add a comparatively large number of objects to a relationship, use a variation of the technique given in Scenario 1. In this variation:

1. Obtain an OREF (*ObjA*) for Class A.
2. Obtain the ID for an instance of ClassB.
3. Use the property setter method of the relationship property of *ObjA*, passing the ID as the argument.

   If the relationship property is named *MyRel*, the property setter method is named **MyRelSetObjectId()**.

   (For details on property setter methods, see the chapter "Using and Overriding Property Methods."

Consider the example classes described in Scenario 1. For those classes, the following steps would insert a large number of invoice items into an invoice (and would do so more rapidly than the technique given in that section):

**ObjectScript**

```
set invoice=##class(MyApp.Invoice).%New()
//set some properties such as the customer name and invoice date
do invoice.%Save()
set id=invoice.%Id()
kill invoice  //OREF is no longer needed

for index = 1:1:(1000)
 {
   set Item=##class(MyApp.LineItem).%New()
   //set properties of the invoice item

   //connect to the invoice
   do Item.InvoiceSetObjectId(id)
   do Item.%Save()
 }
```

# 14.5 Removing a Relationship

In the case of a one-to-many relationship, it is possible to remove a relationship between two objects. One way to do so is as follows:

1. Open the instance of the child object (or the object on the many side).
2. Set the applicable property of this object equal to null.

For example, there is a one-to-many relationship between Sample.Company and Sample.Employee in the SAMPLES namespace. The following demonstrates that the employee whose ID is 101 works for the company whose ID is 5. Notice that this company has four employees:

```
SAMPLES>set e=##class(Sample.Employee).%OpenId(101)

SAMPLES>w e.Company.%Id()
5
SAMPLES>set c=##class(Sample.Company).%OpenId(5)

SAMPLES>w c.Employees.Count()
4
```

Next for this employee, we set the Company property equal to null. Notice that this company now has three employees:

```
SAMPLES>set e.Company=""

SAMPLES>w c.Employees.Count()
3
```

It is also possible to remove the relationship by modifying the other object. In this case, we use the **RemoveAt()** method of the collection property. For example, the following demonstrates that for the company whose ID is 17, the first employee is employee ID 102:

```
SAMPLES>set e=##class(Sample.Employee).%OpenId(102)

SAMPLES>w e.Company.%Id()
17
SAMPLES>set c=##class(Sample.Company).%OpenId(17)

SAMPLES>w c.Employees.Count()
4
SAMPLES>w c.Employees.GetAt(1).%Id()
102
```

To remove the relationship between this company and this employee, we use the **RemoveAt()** method, passing the value 1 as the argument, to remove the first collection item. Notice that after we do so, this company has three employees:

```
SAMPLES>do c.Employees.RemoveAt(1)

SAMPLES>w c.Employees.Count()
3
```

In the case of a parent-child relationship, it is *not* possible to remove a relationship between two objects. You can, however, delete a child object.

# 14.6 Deleting Objects in Relationships

For a one-to-many relationship, the following rules govern what occurs when you attempt to delete objects:

- The relationship prevents you from deleting an object on the one side, if there are any objects on the many side that reference this object. For example, if you try to delete a company, and the employee table has records that point to that company, the delete operation fails.

  Thus it is necessary to first delete the records on the many side.
- The relationship does not prevent you from deleting an object on the many side (the employee table).

For a parent-child relationship, the rules are different:

- The relationship causes a deletion on the parent side to affect the child side. Specifically, if you delete an object on the parent side, the associated objects on the child side are automatically deleted.

  For example, if there is a parent-child relationship between invoices and line items, if you delete an invoice, its line items are deleted.
- The relationship does not prevent you from deleting an object on the child side (the line item table).

For both one-to-many and parent-child relationships, you can change the default behavior of deleting an object on the one side or the parent side by using the OnDelete property keyword.

# 14.7 Working with Relationships

Relationships are properties. Relationships with a cardinality of one or parent behave like atomic (non-collection) reference properties. Relationships with a cardinality of many or children are instances of the %RelationshipObject class, which has an array-like interface.

For example, you could use the Company and Employee objects defined above in the following way:

### ObjectScript

```
// create a new instance of Company
Set company = ##class(MyApp.Company).%New()
Set company.Name = "Chiaroscuro LLC"

// create a new instance of Employee
Set emp = ##class(MyApp.Employee).%New()
Set emp.LastName = "Weiss"
Set emp.FirstName = "Melanie"

// Now associate Employee with Company
Set emp.Employer = company

// Save the Company (this will save emp as well)
Do company.%Save()

// Close the newly created objects
Set company = ""
Set emp = ""
```

Relationships are fully bidirectional in memory; any operation on either side is immediately visible on the other side. Hence, the code above is equivalent to the following, which instead operates on the company:

### ObjectScript

```
Do company.Employees.Insert(emp)

Write emp.Employer.Name
// this will print out "Chiaroscuro LLC"
```

You can load relationships from disk and use them as you would any other property. When you refer to a related object from the one side, the related object is automatically swizzled into memory in the same way as a reference (object-valued) property. When you refer to a related object from the many side, the related objects are not swizzled immediately; instead a transient %RelationshipObject collection object is created. As soon as any methods are called on this collection, it builds a list containing the ID values of the objects within the relationship. It is only when you refer to one of the objects within this collection that the actual related object is swizzled into memory.

Here is an example that displays all Employee objects related to a specific Company:

### ObjectScript

```
// open an instance of Company
Set company = ##class(Company).%OpenId(id)

// iterate over the employees; print their names
Set key = ""

Do {
   Set employee = company.Employees.GetNext(.key)
   If (employee '= "") {
       Write employee.Name,!
   }
} While (key '= "")
```

In this example, closing *company* removes the Company object and all of its related Employee objects from memory. Note, however, that every Employee object contained in the relationship will be swizzled into memory by the time the loop completes. To reduce the amount of memory that this operation uses—perhaps there are thousands of Employee objects—then modify the loop to "unswizzle" the Employee object after displaying the name, by calling the **%UnSwizzleAt()** method:

### ObjectScript

```
Do {
   Set employee = company.Employees.GetNext(.key)
   If (employee '= "") {
       Write employee.Name,!
       // remove employee from memory
       Do company.Employees.%UnSwizzleAt(key)
   }
} While (key '= "")
```

**Important:** Relationships do not support the list interface. That means you cannot get the count of related objects and iterate over the relationship by incrementing a pointer from one (1) by one (1) up to the number of related objects; instead, you must use array-collection style iteration. For more information on iterating through objects in a relationship, see the reference page for %Library.RelationshipObject.

# 14.8 SQL Projection of Relationships

As described earlier in this book, a persistent class is projected as an SQL table. This section describes how relationships of such a class are projected to SQL.

**Note:** Although you can modify the projection of the other properties of the classes involved, it is not possible to modify the SQL projection of relationships per se. For example, it is not supported to specify the *CLASSNAME* property parameter for the relationship. This parameter is mentioned in "Defining Object-Valued Properties" earlier in this book.

## 14.8.1 SQL Projection of One-to-Many Relationships

This section describes the SQL projection of a one-to-many relationship. As an example, consider the example one-to-many classes shown earlier. In this case, the classes are projected as follows:

- On the one side (that is, in the company class), there is no field that represents the relationship. The company table has fields for other properties, but there is no field that holds the employees.
- On the many side (that is, in the employee class), the relationship is a simple reference property, and that is projected to SQL in the same way as other reference properties. The employee table has a field named `Employer`, which points to the company table.

To query these tables together, you can query the employee table and use arrow syntax, as in the following example:

**SQL**

```
SELECT Employer->Name, LastName,FirstName FROM MyApp.Employee
```

Or you can perform an explicit join, as in the following example:

**SQL**

```
SELECT c.Name, e.LastName, e.FirstName FROM MyApp.Company c, MyApp.Employee e WHERE e.Employer = c.ID
```

Also, this pair of relationship properties implicitly adds a foreign key to the employee table; the foreign key has UPDATE and DELETE both specified as NOACTION.

## 14.8.2 SQL Projection of Parent-Child Relationships

Similarly, consider the example parent-child classes shown earlier, which have a parent-child relationship between an invoice and its line items. In this case, the classes are projected as follows:

- On the parent side (that is, in the invoice class), there is no field that represents the relationship. The invoice table has fields for other properties, but there is no field that holds the line items.

- On the child side (that is, in the line item class), the relationship is a simple reference property, and that is projected to SQL in the same way as other reference properties. The line item table has a field named `Invoice`, which points to the invoice table.
- Also on the child side, the IDs always include the ID of the parent record, even if you explicitly attempt to create an IDKey based exclusively on the child. Also, if the definition of the IDKey in the child class explicitly includes the parent relationship, the compiler recognizes this and does not add it again; this allows you to alter the sequence in which the parent reference appears as a subscript in the generated global references.

  As a consequence, it is not possible to add a bitmap index to this property, although other forms of index are permitted.

To query these tables together, you can query the invoice table and use arrow syntax, as in the following example:

**SQL**

```
SELECT
Invoice->Buyer, Invoice->InvoiceDate, ID, ProductSKU, UnitPrice
FROM MyApp.LineItem
```

Or you can perform an explicit join, as in the following example:

**SQL**

```
SELECT
i.Buyer, i.InvoiceDate, l.ProductSKU,l.UnitPrice
FROM MyApp.Invoice i, MyApp.LineItem l
WHERE i.ID = l.Invoice
```

Also, for the class on the child side, the projected table is “adopted” as a child table of the other table.

# 14.9 Creating Many-to-Many Relationships

Caché does not directly support many-to-many relationships, but this section describes how to model such a relationship indirectly.

To establish a many-to-many relationship between class A and class B, do the following:

1. Create a intermediate class that will define each relationship.
2. Define a one-to-many relationship between that class and class A.
3. Define a one-to-many relationship between that class and class B.

Then, create a record in the intermediate class for each relationship between an instance of class A and an instance of class B.

For example, suppose that class A defines doctors; this class defines the properties Name and Specialty. Class B defines patients; this class defines the properties Name and Address. To model the many-to-many relationship between doctors and patients, we could define an intermediate class as follows:

### Class Definition

```
/// Bridge class between MN.Doctor and MN.Patient
Class MN.DoctorPatient Extends %Persistent
{

Relationship Doctor As MN.Doctor [ Cardinality = one, Inverse = Bridge ];

Index DoctorIndex On Doctor;

Relationship Patient As MN.Patient [ Cardinality = one, Inverse = Bridge ];

Index PatientIndex On Patient;
}
```

Then the doctor class looks like this:

### Class Definition

```
Class MN.Doctor Extends %Persistent
{

Property Name;

Property Specialty;

Relationship Bridge As MN.DoctorPatient [ Cardinality = many, Inverse = Doctor ];

}
```

And the patient class looks like this:

### Class Definition

```
Class MN.Patient Extends %Persistent
{

Property Name;

Property Address;

Relationship Bridge As MN.DoctorPatient [ Cardinality = many, Inverse = Patient ];

}
```

The easiest way to query both doctors and patients is to query the intermediate table. The following shows an example:

```
SELECT top 20 Doctor->Name as Doctor, Doctor->Specialty, Patient->Name as Patient
FROM MN.DoctorPatient order by doctor

Doctor  Specialty       Patient
Davis,Joshua M. Dermatologist   Wilson,Josephine J.
Davis,Joshua M. Dermatologist   LaRocca,William O.
Davis,Joshua M. Dermatologist   Dunlap,Joe K.
Davis,Joshua M. Dermatologist   Rotterman,Edward T.
Davis,Joshua M. Dermatologist   Gibbs,Keith W.
Davis,Joshua M. Dermatologist   Black,Charlotte P.
Davis,Joshua M. Dermatologist   Dunlap,Joe K.
Davis,Joshua M. Dermatologist   Rotterman,Edward T.
Li,Umberto R.   Internist       Smith,Wolfgang J.
Li,Umberto R.   Internist       Ulman,Mo O.
Li,Umberto R.   Internist       Gibbs,Keith W.
Li,Umberto R.   Internist       Dunlap,Joe K.
Quixote,William Q.      Surgeon Black,Charlotte P.
Quixote,William Q.      Surgeon LaRocca,William O.
Quixote,William Q.      Surgeon Black,Charlotte P.
Quixote,William Q.      Surgeon Smith,Wolfgang J.
Quixote,William Q.      Surgeon LaRocca,William O.
Quixote,William Q.      Surgeon LaRocca,William O.
Quixote,William Q.      Surgeon Black,Charlotte P.
Salm,Jocelyn Q. Allergist       Tsatsulin,Mark S.
```

As a variation, you can use a parent-child relationship in place of one of the one-to-many relationships. This provides the physical clustering of the data as described earlier in this chapter, but it means that you cannot use a bitmap index on that relationship.

## 14.9.1 Variation with Foreign Keys

Rather than defining relationships between the intermediate class and classes A and B, you can use reference properties and foreign keys, so that the intermediate class `MN.DoctorPatient` looks like this instead of the version shown previously:

### Class Definition

```
Class MN.DoctorPatient Extends %Persistent
{

Property Doctor As MN.Doctor;

ForeignKey DoctorFK(Doctor) References MN.Doctor();

Property Patient As MN.Patient;

ForeignKey PatientFK(Patient) References MN.Patient();

}
```

Foreign keys are discussed in more detail in "Using Foreign Keys" in *Using Caché SQL*. Also see "Foreign Key Definitions" in the reference "Class Definitions" in the *Class Definition Reference*.

One advantage to using a simple foreign key model is that no inadvertent swizzling of large numbers of objects will occur. One disadvantage is that no automatic swizzling is available.

# 15

# Other Options for Persistent Classes

This chapter describes other options that are available for persistent classes. It discusses the following topics:

- How to define a read-only class
- How to add indices
- How to add foreign keys
- How to add triggers
- How to refer to fields from ObjectScript
- How to add row-level security

Also see the chapters "Introduction to Persistent Objects", "Working with Persistent Objects", and "Defining Persistent Classes," as well as the appendix "Using the Object Synchronization Feature."

When viewing this book online, use the preface of this book to quickly find other topics.

## 15.1 Defining a Read-Only Class

It is possible to define a persistent class whose objects can be opened but not saved or deleted. To do this, specify the *READONLY* parameter for the class as 1:

### Class Member

```
Parameter READONLY = 1;
```

This is only useful for cases where you have objects that are mapped to preexisting storage (such as existing globals or an external database). If you call the **%Save()** method on a read-only object, it will always return an error code.

## 15.2 Adding Indices

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 automatically span the entire extent of the class in which they are defined. If a Person class has a subclass Student, all indices defined in Person contain both Person objects and Student objects. Indices defined in the Student class contain only Student objects.

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. (See the Data keyword below.)

For additional information on indices, refer to the "Defining and Building Indices" chapter in *Caché SQL Optimization Guide*; of particular interest may be the section "Properties That Can Be Indexed." Also see "Index Definitions" in the *Caché Class Definition Reference*.

# 15.3 Adding Foreign Keys

To enforce referential integrity between tables you can define foreign keys in the corresponding persistent classes. When a table containing a foreign key constraint is modified, the foreign key constraints are checked. One way to add foreign keys is to add relationships between classes; see the chapter "Defining and Using Relationships." You can also add explicit foreign keys to classes. For information, see "Using Foreign Keys" in *Using Caché SQL*. Also see "Foreign Key Definitions" in the *Caché Class Definition Reference*.

# 15.4 Adding Triggers

Because Caché SQL supports the use of triggers, any trigger associated with a persistent class is included as part of the SQL projection of the class.

Triggers are code segments executed when specific events occur in Caché SQL. Caché supports triggers based on the execution of **INSERT**, **UPDATE**, and **DELETE** commands. The specified code will be executed either immediately before or immediately after the relevant command is executed, depending on the trigger definition. Each event can have multiple triggers as long as they are assigned an execution order.

If a trigger is defined with `Foreach = row/object`, then the trigger is also called at specific points during object access. See "Triggers and Transactions" in "Using Triggers" in *Using Caché SQL*.

Triggers are also fired by the persistence methods used by the legacy storage class, %CacheSQLStorage because it uses SQL statements internally to implement its persistent behavior.

For more information on triggers, see "Triggers" in *Using Caché SQL*. Also see "Trigger Definitions" in the *Caché Class Definition Reference*.

# 15.5 Referring to Fields from ObjectScript

Within a class definition, there are several places that may include ObjectScript code used in SQL. For example, SQL computed field code and trigger code is executed from within SQL. In these cases, there is no concept of a current object, so it is not possible to use dot syntax to access or set data within a specific instance. Instead, you can access the same data as fields within the current row using field syntax.

To reference a specific field of the current row, use the `{fieldname}` syntax where *fieldname* is the name of the field.

For example, the following code checks if the salary of an employee is less than 50000:

```
If {Salary} < 50000 {
   // actions here...
}
```

**Note:** In UPDATE trigger code, `{fieldname}` denotes the updated field value. In DELETE trigger code, `{fieldname}` denotes the value of the field on disk.

To refer to the current field in a SQL computed field, use the `{*}` syntax.

For example, the following code might appear in the computed code for a Compensation field to compute its value based on the values of Salary and Commission fields:

```
Set {*} = {Salary} + {Commission}
```

For trigger-specific syntax, see the "Special Trigger Syntax" section in the "Defining Triggers" chapter in *Using Caché SQL*.

# 15.6 Adding Row-Level Security

In addition to its general security, Caché offers SQL security with a granularity of a single row. This is called *row-level security*. With row-level security, each row holds a list of authorized viewers, which can be either users or roles. For more information on users and roles, see the "Users" and "Roles" chapters of the *Caché Security Administration Guide*.

Typically, SQL security is controlled by granting **SELECT** privilege on a table or view to a user or role. The use of roles simplifies access control when the number of security roles is substantially fewer than the number of users. In most cases, view-level security provides adequate control over which rows each user can select; however, when the number of views required to achieve the desired control becomes very large, another alternative for fine-grained access control is needed.

For example, a hospital may make patient-specific data available online to each patient. Creating a separate view for each patient is not a practical alternative; instead, fine-grained access control, in conjunction with the Caché role-based authentication model, enables this type of application to be created efficiently and securely through row-level security.

The following are constraints on the use of row-level security:

- Row-level security is only available for persistent classes.
- Row-level security is only available for tables instantiated on the Caché server. It is not available for link tables (that is, those that are instantiated on foreign servers).
- Row-level security is only enforced when accessing rows from SQL. It is not enforced when directly accessing globals or when accessing globals via the object interface.

## 15.6.1 Setting Up Row-Level Security

To enable row-level security for a table, edit the definition of the class from which the table is projected.

1. In the class definition code, set the value of *ROWLEVELSECURITY* to 1, such as:

   ```
   ROWLEVELSECURITY = 1;
   ```

   This definition for the parameter means that row-level security is active and that the class uses the generated %READERLIST property to store information about users and roles with authorized access to the row.

Alternatively, you can define the parameter as follows:

```
ROWLEVELSECURITY = rlsprop;
```

Where *rlsprop* is the name of a property in the same class. This alternative means that row-level security is active and that the class uses the given property to store information about users and roles with authorized access to the row. In this case, also add an index to the class as follows:

```
Index %RLI On rlsprop;
```

2. Define a **%SecurityPolicy()** class method, which determines and specifies the role and usernames that are permitted to select the row, subject to view and table **SELECT** privileges.

   The structure of the **%SecurityPolicy()** method is:

```
ClassMethod %SecurityPolicy() As %String [ SqlProc ]
{
    QUIT ""
}
```

   Its characteristics are:

   - It is a class method with the required name "%SecurityPolicy".
   - It returns a string (type %String).
   - It takes zero or more arguments. If this method takes any arguments, each must match a property name in the class and they must all be distinct from each other.
   - The SqlProc keyword specifies that the method can be invoked as a stored procedure.
   - The **QUIT** statement of the method returns the users or roles that may view the row. If there is more than one user or role, **QUIT** must return a comma-separated list of their names. Returning the null string (as in the example) specifies that the row is visible to all users who hold the **SELECT** privilege on the table.

   **Important:** A user who is assigned to the **%All** role does not automatically have access to rows in a table that are protected with row-level security. If **%All** is to have access to such a row, the **%SecurityPolicy()** method must explicitly specify this.

3. Compile the class and any dependent classes.

## 15.6.2 Adding Row-Level Security to a Table with Existing Data

To add row-level security to a table with existing data, first follow the procedure described in the previous section, "Setting Up Row-Level Security." Then:

1. Rebuild the indices for the table.
2. Update the value of the property that lists the users and roles who can view each row.

### 15.6.2.1 Rebuilding the Indices

**CAUTION:** Do not rebuild indices while users are accessing the data for this table. Doing so may result in inaccurate query results.

The procedure to rebuild the indices for a table is:

1. If the table has any views defined that have the WITH CHECK OPTION clause, remove these views with the DROP VIEW command. (You can re-create these views after updating who has access to each row).

2. From the Management Portal home page, go to the **SQL** page (**System Explorer** > **SQL**) page.
3. Select the namespace that contains the table.
4. Under **Tables**, select the name of the table. This displays the **Catalog Details** for the table.
5. On the **Actions** drop-down list, click **Rebuild Table's Indices**.

For more information on rebuilding indices, see "Defining Indices" in the chapter "Defining and Building Indices" in *Caché SQL Optimization Guide*.

#### 15.6.2.2 Updating Who Can View Each Row

The procedure to do this is:

1. From the Management Portal home page, go to the **SQL** page (**System Explorer** > **SQL**) page.
2. Select the namespace that contains the table.
3. Click **Execute Query**.
4. In the editable area, issue a statement to update the table. It should have the following form:

```
UPDATE MySchema.MyClass SET rlsprop =
               MySchema.SecurityPolicy(MySQLColumnName1, ...)
```

   where

   - *MySchema* is the schema (package) containing the class.
   - *MyClass* is the name of the class.
   - *rlsprop* is the field containing the list of users and roles who can read the row. This is %READERLIST by default and the property name specified in the declaration of the *ROWLEVELSECURITY* parameter otherwise.
   - *SecurityPolicy* is value specified by the SqlName value in the definition of the **%SecurityPolicy()** method. If there is no explicit SQL name for the **%SecurityPolicy()** method and its class is MySchema.MyClass, then its default name is **myClass_sys_SecurityPolicy** (with a fully qualified form of **MySchema.MyClass_sys_SecurityPolicy**).
   - *MySQLColumnName1*, ... is the set of SQL column names corresponding to the arguments, if any, defined in the **%SecurityPolicy()** class method.

5. Click **Execute**.
6. If desired, re-create any view that you initially removed.

### 15.6.3 Performance Tips and Information

The %READERLIST property is a calculated field and its value is determined by the **%SecurityPolicy()** method. Whenever an **INSERT** or **UPDATE** occurs, **%SecurityPolicy()** is invoked for that row and populates the value of %READERLIST.

A collection index on the %READERLIST property is defined, and can be exploited by the query optimizer to minimize the performance impact when row-level security is enabled.

By default, when you set *ROWLEVELSECURITY* equal to 1, a collection index is defined for the %READERLIST property (column) because the security policy can, in general, return more than one comma-separated role or username. If your security policy never returns more than one user or role name, then you can override the *ROWLEVELSECURITY* parameter and explicitly define the %RLI index as an ordinary (non-collection) bitmap index. This generally provides optimal performance.

## 15.6.4 Security Tips and Information

Keep in mind the following security factors when using row-level security:

- Row-level security operates in addition to table-level security. To execute a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement, a user must have been granted both table-level access and row-level access for the relevant row.
- User privileges are checked dynamically at runtime — when there is an attempt to execute an SQL command.
- If you create an updateable view and specify **WITH CHECK OPTION**, then an **INSERT** operation on that view checks if the row to be inserted passes the **WHERE** clause that is specified in the view. Further, if you are creating the view from a table with row-level security, then the **INSERT** fails if either the **WHERE** clause of the view or the implicit row-level security predicate would cause that row to not be visible if you were to issue a command of **SELECT * FROM** on the view.
- If you have access to a row, it is possible to change the value of the %READERLIST field (or whatever field holds the list of users and roles who can view the row). This means that you can perform an action that, directly or indirectly, removes your access to the row.
- If you attempt to insert a row that would have violated a UNIQUE constraint if row-level security had not been defined, then it will still violate the constraint if row-level security is defined, regardless of whether the row causing the constraint failure is visible to the updating transaction.

# 16

# Defining Method and Trigger Generators

A method generator is a specific kind of method that generates its own runtime code. Similarly, a trigger generator is a trigger that generates its own runtime code. This chapter discusses them and covers the following topics:

- Introduction
- Basics
- How generators work
- Values available to method generators
- Values available to trigger generators
- How to define method generators
- Generators and INT code
- Generator methods and subclasses

This chapter primarily discusses method generators, but the details are similar for trigger generators.

Also see the chapter "Defining and Calling Methods" and see "Adding Triggers" in the chapter "Other Options for Persistent Classes."

When viewing this book online, use the preface of this book to quickly find other topics.

## 16.1 Introduction

A powerful feature of Caché is the ability to define method generators: small programs that are invoked by the class compiler to generate the runtime code for a method. Similarly a trigger generator is invoked by the class compiler and generates the runtime code for a trigger.

Method generators are used extensively within the Caché class library. For example, most of the methods of the %Persistent class are implemented as method generators. This makes it possible to give each persistent class customized storage code, instead of less efficient, generic code. Most of the Caché data type class methods are also implemented as method generators. Again, this gives these classes the ability to provide custom implementations that depend on the context in which they are used.

You can use method and trigger generators within your own applications. For method generators, a common usage is to define one or more utility superclasses that provide specialized methods for the subclasses that use them. The method generators within these utility classes create special code based on the definition (properties, methods, parameter values,

etc.) of the class that uses them. Good examples of this technique are the %Populate and %XML.Adaptor classes provided within the Caché library.

# 16.2 Basics

A method generator is simply a method of a Caché class that has its CodeMode keyword set to "objectgenerator":

### Class Definition

```
Class MyApp.MyClass Extends %RegisteredObject
{
Method MyMethod() [ CodeMode = objectgenerator ]
    {
        Do %code.WriteLine(" Write """ _ %class.Name _ """")
        Do %code.WriteLine(" Quit")
        Quit $$$OK
    }
}
```

When the class MyApp.MyClass is compiled, it ends up with a **MyMethod** method with the following implementation:

### ObjectScript

```
 Write "MyApp.MyClass"
 Quit
```

**Note:** The value of CodeMode in the previous example is "objectgenerator", since this method generator uses the preferred, object-based, method generator mechanism. Prior to version 5 of Caché, there was a different preferred mechanism, in which the value of CodeMode was "generator". While the older mechanism is preserved for compatibility, new applications should use "objectgenerator".

You can also define trigger generators. To do so, use CodeMode = "objectgenerator" in the definition of a trigger. The values available within your trigger are slightly different than those in a method generator.

# 16.3 How Generators Work

A method generator takes effect when you compile a class. The operation of a method generator is straightforward. When you compile a class definition, the class compiler does the following:

1. It resolves inheritance for the class (builds a list of all inherited members).
2. It makes a list of all methods specified as method generators (by looking at the CodeMode keyword of each method).
3. It gathers the code from all method generators, copies it into one or more temporary routines, and compiles them (this makes it possible to execute the method generator code).
4. It creates a set of transient objects that represent the definition of the class being compiled. These objects are made available to the method generator code.
5. It executes the code for every method generator.

   If present, the compiler will arrange the order in which it invokes the method generators by looking at the value of the GenerateAfter keyword for each of the methods. This keyword gives you some control in cases where there may be compiler timing dependencies among methods.

6. It copies the results of each method generator (lines of code plus any changes to other method keywords) into the compiled class structure (used to generate the actual code for the class).

   Note that the original method signature (arguments and return type), as well as any method keyword values, are used for the generated method. If you specify a method generator as having a return type of %Integer, then the actual method will have a return type of %Integer.

7. It generates the executable code for the class by combining the code generated by the method generators along with the code from all the non-method generator methods.

The details are similar for trigger generators.

# 16.4 Values Available to Method Generators

The key to implementing method generators is understanding the context in which method generator code is executed. As described in the previous section, the class compiler invokes the method generator code at the point after it has resolved class inheritance but before it has generated code for the class. When it invokes method generator code, the class compiler makes the following variables available to the method generator code:

***Table 16–1: Variables Available to Method Generators***

| Variable | Description |
|---|---|
| *%code* | An instance of the %Stream.MethodGenerator class. This is a stream into which you write the code for the method. |
| *%class* | An instance of the %Dictionary.ClassDefinition class. It contains the original definition of the class. |
| *%method* | An instance of the %Dictionary.MethodDefinition class. It contains the original definition of the method. |
| *%compiledclass* | An instance of the %Dictionary.CompiledClass class. It contains the *compiled* definition of the class being compiled. Thus, it contains information about the class after inheritance has been resolved (such as the list of all properties and methods, including those inherited from superclasses). |
| *%compiledmethod* or *%objcompiledmethod* | An instance of the %Dictionary class for the compiled method, for example,%Dictionary.CompiledMethod, %Dictionary.CompiledPropertyMethod or%Dictionary.CompiledIndexMethod. It contains the *compiled* definition of the method being generated. |
| *%parameter* | An array that contains the values of any class parameters indexed by parameter name. For example, *%parameter("MYPARAM")*, contains the value of the *MYPARAM* class parameter for the current class. This variable is provided as an easier alternative to using the list of parameter definitions available via the *%class* object. |
| *%kind* or *%membertype* | For member methods, the kind of class member that relates to this method, for example, a for property methods or `i` for index methods. |
| *%mode* | The type of method, for example, `method`, `propertymethod`, or `indexmethod`. |
| *%pqname* or *%member* | For member methods, the name of the class member that relates to this method. |

# 16.5 Values Available to Trigger Generators

Like methods, triggers can be defined as generators. That is, you can use CodeMode = "objectgenerator" in the definition of a trigger. The following variables are available within the trigger generator:

***Table 16–2: Added Variables Available to Trigger Generators***

| Variable | Description |
|---|---|
| *%code*, *%class*, *%compiledclass*, and *%parameter* | See the preceding section. |
| *%trigger* | An instance of the %Dictionary.TriggerDefinition class. It contains the original definition of the trigger. |
| *%compiledtrigger* or *%objcompiledmethod* | An instance of the %Dictionary.CompiledTrigger class. It contains the *compiled* definition of the trigger being generated. |
| *%kind* or *%membertype* | For triggers, this is the value `t`. |
| *%mode* | For triggers, this is the value `trigger`. |
| *%pqname* or *%member* | The name of this trigger. |

# 16.6 Defining Method Generators

To define a method generator, do the following:

1. Define a method and set its CodeMode keyword to "objectgenerator".
2. In the body of the method, write code that generates the actual method code when the class is compiled. This code uses the *%code* object to write out the code. It will most likely use the other available objects as inputs to decide what code to generate.

The following is an example of a method generator that creates a method that lists the names of all the properties of the class it belongs to:

### Class Member

```
ClassMethod ListProperties() [ CodeMode = objectgenerator ]
{
    For i = 1:1:%compiledclass.Properties.Count() {
        Set prop = %compiledclass.Properties.GetAt(i).Name
        Do %code.WriteLine(" Write """ _ prop _ """,!")
    }
    Do %code.WriteLine(" Quit")
    Quit $$$OK
}
```

This generator will create a method with an implementation similar to:

#### ObjectScript

```
 Write "Name",!
 Write "SSN",!
 Quit
```

Note the following about the method generator code:

1. It uses the **WriteLine** method of the *%code* object to write lines of code to a stream containing the actual implementation for the method. (You can also use the **Write** method to write text without an end-of-line character).
2. Each line of generated code has a leading space character. This is required because ObjectScript does not allow commands within the first space of a line. This would not be the case if our method generator is creating Basic or Java code.
3. As the lines of generated code appear within strings, you have to be very careful about escaping quotation mark characters by doubling them up ("").
4. To find the list of properties for the class, it uses the *%compiledclass* object. It could use the *%class* object, but then it would only list properties defined within the class being compiled; it would not list inherited properties.
5. It returns a status code of $$$OK, indicating that the *method generator* ran successfully. This return value has nothing to do with the actual implementation of the method.

### 16.6.1 Method Generators for Other Languages

You can generate code for different languages. To do so, set the Language property of the *%code* object to specify the target language.

By default, the language for the generated code is the same as the language used to write the code generator method (specified by the Language keyword).

### 16.6.2 Specifying CodeMode within a Method Generator

By default, a method generator will create a "code" method (that is, the CodeMode keyword for the generated method is set to "code"). You can change this using the CodeMode property of the *%code* object.

For example, the following method generator will generate an ObjectScript expression method:

#### Class Member

```
Method Double(%val As %Integer) As %Integer [ CodeMode = objectgenerator ]
{
    Set %code.CodeMode = "expression"
    Do %code.WriteLine("%val * 2")
}
```

## 16.7 Generators and INT Code

For method and trigger generators, it can be very useful to display the corresponding INT code after compiling the class. See "Displaying INT Code" in the chapter "Useful Skills to Learn" in *Caché Programming Orientation Guide*.

Note that if the generator is simple enough to be implemented in the kernel, there is no generated .INT code for it.

# 16.8 Generator Methods and Subclasses

This section discusses topics specific to generator methods in subclasses of the class in which they were defined.

It is necessary, of course, to compile any subclasses after compiling the superclass.

## 16.8.1 Method Regeneration in Subclasses

When you subclass a class that defines generator methods, Caché uses the same compilation rules that are described earlier in this chapter. Caché does not, however, recompile a method in a subclass if the generated code looks the same as the superclass generated code. This logic does not consider whether the include files are the same for both classes. If the method uses a macro that is defined in an include file and if the subclass uses a different include file, Caché would not recompile the method in the subclass. You can, however, force the generator method to be recompiled in every class. To do so, specify the method keyword ForceGenerate for that method. There may be additional scenarios where this keyword is needed.

## 16.8.2 Invoking the Method in the Superclass

If you need a subclass to use the method generated for the superclass, rather than a locally generated method, do the following in the subclass: define the generator method so that it just returns `$$$OK`, as in the following example:

### Class Member

```
ClassMethod Demo1() [ CodeMode = objectgenerator ]
{
    quit $$$OK
}
```

## 16.8.3 Removing a Generated Method

You can remove a generated method from a subclass, so that it cannot be invoked in that class. To do so, when you define the generator method in the superclass, include logic that examines the name of the current class and generates code only in the desired scenarios. For example:

```
ClassMethod Demo3() [ CodeMode = objectgenerator ]
{
    if %class.Name="RemovingMethod.ClassA" {
        Do %code.WriteLine(" Write !,""Hello from class: " _ %class.Name _ """")
    }
    quit $$$OK
}
```

If you try to invoke this method in any subclass, you receive the error `<METHOD DOES NOT EXIST>`.

Note that this logic is subtly different from that described in the previous section. If a generator method in a given class exists but has a null implementation, the method of the superclass, if any, is used instead. But if a generator method in a given class does not generate code for a given subclass, the method does not exist in that subclass and cannot be invoked.

# 17

# Defining and Using Class Queries

This chapter discusses *class queries*, which act as named queries that are part of a class structure and that can be accessed via dynamic SQL. It discusses the following topics:

- Introduction
- How to use class queries
- How to define basic class queries
- How to define custom class queries
- How to define parameters for custom queries
- Additional custom class example
- Uses of custom queries
- SQL cursors and class queries

When viewing this book online, use the preface of this book to quickly find related topics.

## 17.1 Introduction to Class Queries

A class query is a tool — contained in a class and meant for use with dynamic SQL — to look up records that meet specified criteria. With class queries, you can create predefined lookups for your application. For example, you can look up records by name, or provide a list of records that meet a particular set of conditions, such as all the flights from Paris to Madrid.

By creating a class query, you can avoid having to look up a particular object by its internal ID. Instead, you can create a query that looks based on any class properties that you want. These can even be specified from user input at runtime.

If you define a custom class query, your lookup logic can use ObjectScript and can be arbitrarily complex.

There are two kinds of class queries:

- Basic class queries, which use the class %SQLQuery and an SQL SELECT statement.
- Custom class queries, which use the class %Query and custom logic to execute, fetch, and close the query.

Note that you can define class queries within *any* class; there is no requirement to contain them within persistent classes.

**Important:** Do not define a class query that depends upon the results of another class query. Such a dependency is not supported.

# 17.2 Using Class Queries

Before looking at how to define class queries, it is useful to see how you can use them. In server-side code, you can use a class query as follows:

1. Use **%New()** to create an instance of %SQL.Statement.
2. Call the **%PrepareClassQuery()** method of that instance. As arguments, use the following, in order:
   a. Fully qualified name of the class that defines the query that you want to use.
   b. Name of the query in that class.

   This method returns a %Status value, which you should check.
3. Call the **%Execute()** method of the %SQL.Statement instance. This returns an instance of %SQL.StatementResult.
4. Use methods of %SQL.StatementResult to retrieve data from the result set. For details, see "Dynamic SQL" in *Using Caché SQL*.

Note that you can use the older dynamic SQL API (%ResultSet) in a similar manner.

The following shows a simple example that you can use in the SAMPLES namespace. This example uses the **ByName** query of Sample.Person:

### ObjectScript

```
 // classquerydemo

#include %occInclude

 set statement=##class(%SQL.Statement).%New()
 set status=statement.%PrepareClassQuery("Sample.Person","ByName")
 if $$$ISERR(status) { do $system.OBJ.DisplayError(status) }
 set resultset=statement.%Execute()
 while resultset.%Next() {
    write !, resultset.%Get("Name")
 }
```

If you are using the Caché Java or ActiveX binding, you can use the result set classes that are part of that binding.

If the query is marked with SqlProc, which defines it as an ODBC or JDBC stored procedure, you can invoke it as a stored procedure from an SQL context. See "Defining and Using Stored Procedures" in *Using Caché SQL*.

# 17.3 Defining Basic Class Queries

To define a basic class query, define a query as follows:

- (For simple class queries) The type should be %SQLQuery.
- In the argument list, specify any arguments that the query should accept.
- In the body of the definition, write an SQL SELECT statement.

  In this statement, to refer to an argument, precede the argument name with a colon (:).

  This SELECT statement should not include an INTO clause.

- Specify the *ROWSPEC* parameter of the query (in parentheses, after the query type). This parameter provides information on the names, data types, headings, and order of the fields in each row of the result set of the query. The second subsection provides the details.
- Optionally specify the *CONTAINID* parameter of the query (in parentheses, after the query type). This parameter specifies the column number of the field, if any, that contains the ID for a particular row; the default is 1. The third subsection provides the details.

  Together, the *ROWSPEC* and *CONTAINID* parameters are known as the *query specification*.
- Include the SqlProc keyword in the query definition.

  You can omit this step if you plan to use %ResultSet to invoke the query and if you do not need to invoke the query as a stored procedure. If you plan to use %SQL.Statement to invoke the query, you *must* specify the SqlProc keyword.
- Optionally specify the SqlName keyword in the query definition, if you want the name of the stored procedure to be other than the default name.

  These are compiler keywords, so include them in square brackets after any parameters, after the query type (%SQLQuery).

Studio provides a wizard (the New Query Wizard) that you can use to define such a basic class query. The following subsection shows an example.

## 17.3.1 Example

The following shows a simple example:

### Class Member

```
Query ListEmployees(City As %String = "")
   As %SQLQuery (ROWSPEC="ID:%Integer,Name:%String,Title:%String", CONTAINID = 1) [SqlProc,
SqlName=MyProcedureName]
{
SELECT ID,Name,Title FROM Employee
 WHERE (Home_City %STARTSWITH :City)
 ORDER BY Name
}
```

**Note:** If you call a class query using ADO.NET, ODBC, or JDBC, any string parameters will be truncated to 50 characters by default. To increase the maximum string length for a parameter, specify a *MAXLEN* in the signature, as in the following example:

```
Query MyQuery(MyParm As %String(MAXLEN = 200)) As %SQLQuery [SqlProc]
```

This truncation does not occur if you call the query from the Management Portal or from ObjectScript.

## 17.3.2 About ROWSPEC

The *ROWSPEC* parameter for a query provides information on the names, data types, headings, and order of the fields in each row. It is a quoted and comma-separated list of variable names and data types of the form:

```
ROWSPEC = "Var1:%Type1,Var2:%Type2[:OptionalDescription],Var3"
```

The *ROWSPEC* specifies the order of fields as a comma-separated list. The information for each field consists of a colon-separated list of its name, its data type (if it is different than the data type of the corresponding property), and an optional heading. To edit *ROWSPEC*, the options are:

- Edit the code directly.

- For an already existing query, display the query in the Studio **Inspector** window, expand its list of parameters, and use the available dialog box.

The number of elements in the *ROWSPEC* parameter must match the number of fields in the query. Otherwise, Caché returns a "Cardinality Mismatch" error.

For an example, in the SAMPLES database, the **ByName** query of the Sample.Person sample class is as follows:

#### Class Member

```
Query ByName(name As %String = "")
    As %SQLQuery(CONTAINID = 1, ROWSPEC = "ID:%Integer,Name,DOB,SSN", SELECTMODE = "RUNTIME")
   [ SqlName = SP_Sample_By_Name, SqlProc ]
{
        SELECT ID, Name, DOB, SSN
            FROM Sample.Person
            WHERE (Name %STARTSWITH :name)
            ORDER BY Name
}
```

Here, the *CONTAINID* parameter specifies that the row ID is the first field (the default); note that the first field specified in the SELECT statement is *ID*. The *ROWSPEC* parameter specifies that the fields are *ID* (treated as an integer), *Name*, *DOB*, and *SSN*; similarly, the SELECT statement contains the fields *ID*, *Name*, *DOB*, and *SSN*, in that order.

### 17.3.3 About CONTAINID

*CONTAINID* should be set to the number of the column returning the ID (1, by default) or to 0 if no column returns the ID. If you create a query using the **New Query Wizard**, then Studio automatically assigns the appropriate value to *CONTAINID*, based on the order you specify in that wizard.

**Note:** Caché does not validate the value of *CONTAINID*. If you specify a non-valid value for this parameter, Caché does not throw an error. This means that if your query processing logic depends on this information, you may experience inconsistencies if the *CONTAINID* parameter is set improperly.

### 17.3.4 Other Parameters of the Query Class

In addition to *ROWSPEC* and *CONTAINID*, you can specify the following parameters of the query. These are class parameters for %SQLQuery:

- *SELECTMODE*
- *COMPILEMODE*

For details, see the class reference for %Library.SQLQuery and %Library.Query (its superclass).

## 17.4 Defining Custom Class Queries

Although simple %SQLQuery queries perform all result set management for you, this is not sufficient for certain applications. For such situations, Caché allows you to write *custom queries*, which are defined in methods (which by default are written in ObjectScript). To define a custom query, use the instructions given earlier in this chapter, with the following changes:

- Specify %Query for the query type.
- Leave the body of the query definition empty. For example:

### Class Member

```
Query All() As %Query(CONTAINID = 1, ROWSPEC = "Title:%String,Author:%String")
{
}
```

- Define the following class methods in the same class:
  - *queryname***Execute** — This method must perform any one-time setup.
  - *queryname***Fetch** — This method must return a row of the result set; each subsequent call returns the next row.
  - *queryname***Close** — This method must perform any cleanup operations.

  Where *queryname* is the name of the query.

  Each of these methods accepts an argument (*qHandle*), which is passed by reference. You can use this argument to pass information among these methods.

  These methods define the query. The following subsections provide details on them.

For basic demonstration purposes, the first three subsections show a simple example that *could* also be implemented as a basic class query; you can use this sample in the SAMPLES namespace. These methods implement the code for the following query:

### Class Member

```
Query AllPersons() As %Query(ROWSPEC = "ID:%String,Name:%String,DOB:%String,SSN:%String")
{
}
```

The next section shows a more complex example. Also see "Uses of Custom Queries," for information on other use cases.

## 17.4.1 Defining the querynameExecute() Method

The **querynameExecute()** method must provide all the setup logic needed. The name of the method must be *queryname***Execute**, where *queryname* is the name of the query. This method must have the following signature:

```
ClassMethod queryNameExecute(ByRef qHandle As %Binary,
                             additional_arguments) As %Status
```

Where:

- *qHandle* is used to communicate with the other methods that implement this query.

  This method should set *qHandle* as needed by the *queryname***Fetch** method.

  Although *qHandle* is formally of type %Binary, it can hold any value, including an OREF or a multidimensional array.
- *additional_arguments* is any runtime parameters that the query can use.

Within this implementation of method, use the following general logic:

1. Perform any one-time setup steps.

   For queries using SQL code, this method typically includes declaring and opening a cursor.
2. Set *qHandle* as needed by the *queryname***Fetch** method.
3. Return a status value.

The following shows a simple example, the **AllPersonsExecute()** method for the **AllPersons** query introduced earlier:

### Class Member

```
ClassMethod AllPersonsExecute(ByRef qHandle As %Binary) As %Status
{
    set statement=##class(%SQL.Statement).%New()
    set status=statement.%PrepareClassQuery("Sample.Person","ByName")
    if $$$ISERR(status) { quit status }
    set resultset=statement.%Execute()
    set qHandle=resultset
    Quit $$$OK
}
```

In this scenario, the method sets *qHandle* equal to an OREF, specifically an instance of %SQL.StatementResult, which is the value returned by the **%Execute()** method. This is only one possibility; see "Additional Custom Class Example" for another approach.

As noted earlier, this class query could also be implemented as a basic class query rather than a custom class query. Some custom class queries do, however, use dynamic SQL as a starting point.

## 17.4.2 Defining the querynameFetch() Method

The **querynameFetch()** method must return a single row of data in **$List** format. The name of the method must be *queryname***Fetch**, where *queryname* is the name of the query. This method must have the following signature:

```
ClassMethod queryNameFetch(ByRef qHandle As %Binary,
                           ByRef Row As %List,
                           ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = querynameExecute ]
```

Where:

- *qHandle* is used to communicate with the other methods that implement this query.

  When Caché starts executing this method, *qHandle* has the value established by the *queryname***Execute** method or by the previous invocation (if any) of this method. This method should set *qHandle* as needed by subsequent logic.

  Although *qHandle* is formally of type %Binary, it can hold any value, including an OREF or a multidimensional array.
- *Row* must be either a %List of values representing a row of data being returned or a null string if no data is returned.
- *AtEnd* must be 1 when the last row of data has been reached.
- The PlaceAfter method keyword controls the position of this method in the generated routine code. For *querynameExecute*, substitute the name of the specific **querynameExecute()** method. Be sure to include this if your query uses SQL cursors. (The ability to control this order is an advanced feature that should be used with caution. InterSystems does not recommend general use of this keyword.)

Within this implementation of method, use the following general logic:

1. Check to determine if it should return any more results.
2. If appropriate, retrieve a row of data and create a %List object and place that in the *Row* variable.
3. Set *qHandle* as needed by subsequent invocations (if any) of this method or needed by the **querynameClose()** method.
4. If no more rows exist, set *Row* to a null string and set *AtEnd* to 1.
5. Return a status value.

For the **AllPersons** example, the **AllPersonsFetch()** method could be as follows:

### Class Member

```
ClassMethod AllPersonsFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd As %Integer = 0)
 As %Status
[ PlaceAfter = AllPersonsExecute ]
{
    set rs=$get(qHandle)
    if rs="" quit $$$OK

    if rs.%Next() {
        set Row=$lb(rs.%GetData(1),rs.%GetData(2),rs.%GetData(3),rs.%GetData(4))
        set AtEnd=0
    } else {
        set Row=""
        set AtEnd=1
    }
    Quit $$$OK
}
```

Notice that this method uses the *qHandle* argument, which provides a %SQL.StatementResult object. The method then uses methods of that class to retrieve data. The method builds a $List and places that in the *Row* variable, which is returned as a single row of data. Also notice that the method contains logic to set the *AtEnd* variable when no more data can be retrieved.

As noted earlier, this class query could also be implemented as a basic class query rather than a custom class query. The purpose of this example is to demonstrate setting the *Row* and *AtEnd* variables.

## 17.4.3 The querynameClose() Method

The **querynameClose()** method must perform any needed clean up, after data retrieval has finished. The name of the method must be *queryname***Close**, where *queryname* is the name of the query. This method must have the following signature:

```
ClassMethod queryNameClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = querynameFetch ]
```

Where:

- *qHandle* is used to communicate with the other methods that implement this query.

  When Caché starts executing this method, *qHandle* has the value established by the last invocation of the *queryname***Fetch** method.

- The PlaceAfter method keyword controls the position of this method in the generated routine code. For *querynameFetch*, substitute the name of the specific **querynameFetch()** method. Be sure to include this if your query uses SQL cursors. (The ability to control this order is an advanced feature that should be used with caution. InterSystems does not recommend general use of this keyword.)

Within this implementation of method, remove variables from memory, close any SQL cursors, or perform any other cleanup as needed. The method must return a status value.

For the **AllPersons** example, the **AllPersonsClose()** method could be as follows:

For example, the signature of a **ByNameClose()** method might be:

### Class Member

```
ClassMethod AllPersonsClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = AllPersonsFetch ]
{
        Set qHandle=""
        Quit $$$OK
}
```

## 17.4.4 Generated Methods for Custom Queries

The system automatically generates the **querynameGetInfo()** and **querynameFetchRows()**. Your application does not call any of these methods directly — the %Library.ResultSet object uses them to process query requests.

# 17.5 Defining Parameters for Custom Queries

If the custom query should accept parameters, do the following:

- Include them in the argument list of the query class member. The following example uses a parameter named `MyParm`:

  **Class Member**

  ```
  Query All(MyParm As %String) As %Query(CONTAINID = 1, ROWSPEC = "Title:%String,Author:%String")
  {
  }
  ```

- Include the same parameters in the argument list for *queryname***Execute** method, in the same order as in the query class member.
- In the implementation of the *queryname***Execute** method, use the parameters as appropriate for your needs.

**Note:** If you call a class query using ADO.NET, ODBC, or JDBC, any string parameters will be truncated to 50 characters by default. To increase the maximum string length for a parameter, specify a *MAXLEN* in the signature, as in the following example:

```
Query MyQuery(MyParm As %String(MAXLEN = 200)) As %Query [SqlProc]
```

This truncation does not occur if you call the query from the Management Portal or from ObjectScript.

# 17.6 Additional Custom Query Example

The previous section provides a simple example of a custom class query, one that could easily be implemented instead as a basic class query. *This* section shows a more typical example from the Caché class library. Also see the next section for additional ideas.

**Important:** This example is presented to demonstrate an approach you can use, not to document how the class library implements specific features. Thus this section does not indicate which class currently contains this code, nor will this section be updated to reflect future changes in that class.

In this example, the query builds and uses a process-private global. The query is defined as follows:

**Class Member**

```
Query ByServer() As %Query(ROWSPEC = "Name,Port,PingPort,Renderer,State,StateEx") [ SqlProc ]
{
}
```

The **querynameExecute()** method is as follows:

**Class Member**

```
ClassMethod ByServerExecute(ByRef qHandle As %Binary) As %Status [ Internal ]
{
    Set tSC = $$$OK
    Try {
        Set tRS = ##class(%ResultSet).%New("%ZEN.Report.RenderServer:ByName")
        Kill ^||%ISC.ZRS
        Set tSC = tRS.Execute()
        For {
            Quit:'tRS.Next()
            Set tType = tRS.Get("ServerType")
```

```
            If (tType'=0) && (tType'="") Continue // Not a Render Server
            Set name = tRS.Get("Name")
            Set ^||%ISC.ZRS(name) = $LB(name,tRS.Get("Port"),tRS.Get("PingPort"),tRS.Get("Renderer"))
        }
    }
    Catch (ex) {
        Set tSC = ex.AsStatus()
    }
    Set qHandle = $LB("")
    Quit tSC
}
```

Notice that this method saves the data into a process-private global and not into the *qHandle* variable. Also note that this method uses the older dynamic SQL class (%ResultSet).

The **querynameFetch()** method is as follows:

### Class Member

```
ClassMethod ByServerFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd As %Integer = 0)
As %Status [ Internal, PlaceAfter = ByServerExecute ]
{
    Set index = $List(qHandle,1)
    Set index = $O(^||%ISC.ZRS(index))
    If index="" {
        Set Row = ""
        Set AtEnd = 1
    }
    Else {
        Set Row = ^||%ISC.ZRS(index)
        Set stInt = ..GetState($List(Row,2),$List(Row,3),$List(Row,4))
        Set stExt = $Case(stInt,0:$$$Text("Inactive"),1:$$$Text("Active"),
                    2:$$$Text("Unresponsive"),3:$$$Text("Troubled"),4:$$$Text("Error"),
                    5:$$$Text("Mismatch"),:"")
        Set $List(Row,5) = stInt, $List(Row,6) = stExt
    }
    Set qHandle = $LB(index)
    Quit $$$OK
}
```

Finally, the **querynameClose()** method is as follows:

### Class Member

```
ClassMethod ByServerClose(ByRef qHandle As %Binary) As %Status [ Internal, PlaceAfter = ByServerExecute
 ]
{
    Set qHandle = ""
    Kill ^||%ISC.ZRS
    Quit $$$OK
}
```

# 17.7 When to Use Custom Queries

The following list suggests some scenarios when custom queries are appropriate:

- If it is necessary to use very complex logic to determine whether to include a specific row in the returned data. The **querynameFetch()** method can contain arbitrarily complex logic.
- If you have an API that returns data in format that is inconvenient for your current use case. In such a scenario, you would define the **querynameFetch()** method so that converts data from that format into a $List, as needed by the *Row* variable.
- If the data is stored in a global that does not have a class interface.
- If access to the data requires role escalation. In this scenario, you can perform the role escalation within the **querynameExecute()** method.

- If access to the data requires calling out to the file system (for example, when building a list of files). In this scenario, you can perform the callout within the **querynameExecute()** method and then stash the results either in *qHandle* or in a global.
- If it is necessary to perform a security check, check connections, or perform some other special setup work before retrieving data. You would do such work within the **querynameExecute()** method.

# 17.8 SQL Cursors and Class Queries

If a class query uses an SQL cursor, note the following points:

- Cursors generated from queries of type %SQLQuery automatically have names such as Q14.

  You must ensure that your cursors are given distinct names.
- Error messages refer to the internal cursor name, which typically has an extra digit. Therefore an error message for cursor Q140 probably refers to Q14.
- The class compiler must find a cursor declaration before making any attempt to use the cursor. This means that you must take extra care when defining a custom query that uses cursors.

  The DECLARE statement (usually in **querynameExecute()** method) must be in the same MAC routine as the Close and Fetch and must come before either of them. As shown earlier in this chapter, use the method keyword PlaceAfter in both the **querynameFetch()** and **querynameClose()** method definitions to make sure this happens.

# 18

# Defining and Using XData Blocks

An *XData block* is a class member that consists of a name and a unit of data that you include in a class definition for use by the class after compilation. This chapter discusses XData blocks and covers the following topics:

- Basics
- Example XData Blocks
- Using XData (XML example)
- Using XData (JSON example)

When viewing this book online, use the preface of this book to quickly find related topics.

## 18.1 Basics

An *XData block* is a named unit of data that you include in a class definition, typically for use by a method in the class. Most frequently, it is a well-formed XML document, but it could consist of other forms of data, such as JSON.

You can create an XData block either by typing it directly in Studio or by using a wizard in Studio.

An XData block is a named class member (like properties, methods, and so on). The available XData block keywords include:

- SchemaSpec — Optionally specifies an XML schema against which the XData can be validated.
- XMLNamespace — Optionally specifies the XML namespace to which the XData block belongs. You can also, of course, include namespace declarations within the XData block itself.
- MimeType — The MIME type (more formally, the Internet media type) of the contents of the XData block. The default is `text/xml`.

If used to store XML, contents of the XData block must consist of one root XML element, with any valid contents.

# 18.2 Example XData Blocks

Zen uses XData blocks extensively. These XData blocks are all described in *Using Zen*. 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>
  }
```

# 18.3 Using XData (XML Example)

To access an XML document in an arbitrary XData block programmatically, you use %Dictionary.CompiledXData and other classes in the %Dictionary package.

An XData block is useful if you want to define a small amount of system data. For example, suppose that the `EPI.AllergySeverity` class includes the properties `Code` (for internal use) and `Description` (for display to the users). This class could include an XData block like the following:

### Class Member

```
XData LoadData
{
<table>
 <row>1^Minor</row>
 <row>2^Moderate</row>
 <row>3^Life-threatening</row>
 <row>9^Inactive</row>
 <row>99^Unable to determine</row>
</table>
}
```

The same class could also include a class method that reads this XData block and populates the table, as follows:

### Class Member

```
/// called by EPI.Utils.GenerateData
ClassMethod Setup() As %Status
{
   //first kill extent
   do ..%KillExtent()

   // Get a stream of XML from the XData block contained in this class
   Set xdataID="EPI.AllergySeverity||LoadData"
   Set compiledXdata=##class(%Dictionary.CompiledXData).%OpenId(xdataID)
   Set tStream=compiledXdata.Data
   If '$IsObject(tStream) Set tSC=%objlasterror Quit

   set status=##class(%XML.TextReader).ParseStream(tStream,.textreader)
   //check status
   if $$$ISERR(status) do $System.Status.DisplayError(status) quit

   //iterate through document, node by node
   while textreader.Read()
   {
       if (textreader.NodeType="chars")
       {
```

```
            set value=textreader.Value
            set obj=..%New()
            set obj.Code=$Piece(value,"^",1)
            set obj.Description=$Piece(value,"^",2)
            do obj.%Save()
            }
   }
}
```

Notice the following:

- The XML within the XData is minimal. That it, instead of presenting the allergy severities as XML element with their own elements or attributes, the XData block simply presents rows of data as delimited strings. This approach allows you to write the setup data in a visually compact form.
- The `EPI.AllergySeverity` class is not XML-enabled and does not need to be XML-enabled.

# 18.4 Using XData (JSON Example)

A class could also include an XData block containing JSON content, like the following:

### Class Member

```
XData LoadJSONData [MimeType = "application/json"]
{
   {
      "person":"John",
      "age":30,
      "car":"Ford"
   }
}
```

The same class could also include a class method that reads this XData block and populates a dynamic object, as follows:

### Class Member

```
/// Reads a JSON XData block
ClassMethod SetupJSON() As %Status
{

   // Get a stream of JSON from the XData block contained in this class
   Set xdataID="Demo.XData||LoadJSONData"
   Set compiledXdata=##class(%Dictionary.CompiledXData).%OpenId(xdataID)
   Set tStream=compiledXdata.Data
   If '$IsObject(tStream) Set tSC=%objlasterror Quit

   // Create a dynamic object from the JSON content and write it as a string
   Set dynObject = {}.%FromJSON(tStream)
   Write dynObject.%ToJSON()

}
```

# 19

# Defining Class Projections

This chapter discusses class projections, which provide a way to customize what happens when a class is compiled or removed. It discusses the following topics:

- Introduction
- How to add a projection to a class
- How to define a new projection class

When viewing this book online, use the preface of this book to quickly find related topics.

## 19.1 Introduction

Class projections provide a way to customize what happens when a class is compiled or removed. A class projection associates a class definition with a projection class. The projection class (derived from the %Projection.AbstractProjection class) provides methods that Caché uses to automatically generate additional code at two times:

- When the class is compiled
- When the class is deleted

This mechanism is used by the Java and C++ projections (hence the origin of the term *projection*) to automatically generate the necessary client binding code (Java or C++) whenever a class is compiled.

## 19.2 Adding a Projection to a Class

To add a projection to a class definition, use the Projection statement within a class definition:

### Class Definition

```
class MyApp.Person extends %Persistent
{
Projection JavaClient As %Projection.Java(ROOTDIR="c:\java");
}
```

This example defines a projection named `JavaClient` that will use the %Projection.Java projection class. When the methods of the projection class are called, they will receive the value of the *ROOTDIR* parameter.

A class can have multiple uniquely named projections. In the case of multiple projections, the methods of each projection class will be invoked when a class is compiled or deleted. The order in which multiple projections are handled is undefined.

Caché provides the following projection classes:

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

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

# 19.3 Creating a New Projection Class

To create a new projection class, create a subclass of the %Projection.AbstractProjection class, implement the projection interface methods (see the subsection), and define any needed class parameters. For example:

### Class Definition

```
Class MyApp.MyProjection Extends %Projection.AbstractProjection
{

Parameter MYPARAM;

/// This method is invoked when a class is compiled
ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
    // code here...
    QUIT $$$OK
}

/// This method is invoked when a class is 'uncompiled'
ClassMethod RemoveProjection(cls As %String, ByRef params, recompile as %Boolean) As %Status
{
    // code here...
    QUIT $$$OK
  }
}
```

## 19.3.1 The Projection Interface

Every projection class implements the *projection interface*, a set of methods that are called in response to certain events during the life cycle of a class. This interface consists of the following methods:

### CreateProjection()

The **CreateProjection()** method is a class method that is invoked by the class compiler after it completes the compilation of a class definition. This method is passed the name of the class being compiled as well as an array containing the parameter values (subscripted by parameter name) defined for the projection.

### RemoveProjection()

The **RemoveProjection()** method is a class method that is invoked either:

- When a class definition is deleted
- At the start of a recompilation of the class

This method is passed the name of the class being removed, an array containing the parameter values (subscripted by parameter name) defined for the projection, and a flag indicating whether the method is being called as part of a recompilation or because the class definition is being deleted.

When a class definition containing a projection is compiled, the following events occur:

1. If the class has been compiled previously, it will be *uncompiled* before the new compile begins; that is, all the results of the previous compilation are removed. At this time, the compiler invokes the **RemoveProjection()** method for every projection with a flag indicating that a recompilation is about to occur.

   Note that you cannot call methods of the associated class from within the **RemoveProjection()** method, because the class does not exist at this point.

   Also note that if you add a new projection definition to a class that had been previously compiled (without the projection), then the compiler will call the **RemoveProjection()** method on the next compilation even though the **CreateProjection()** method has never been called. Implementers of the **RemoveProjection()** method must plan for this possibility.

2. After the class is completely compiled (that is, it is ready for use), the compiler will invoke the **CreateProjection()** method for every projection.

When a class definition is deleted, the **RemoveProjection()** method is invoked for every projection with a flag indicating that a deletion has occurred.

# 20

# Defining Callback Methods

Callback methods are called by system methods to allow additional user-supplied processing. Callback methods are identifiable by having names that begin with "%On" or "On", typically followed by the name of the method that invokes them. In most cases, this is the entire name, such as **%OnNew()**.

If a system method has an implemented callback method, then when the system method runs, that method invokes the callback method. For example, **%Delete()** invokes **%OnDelete()**, if **%OnDelete()** is implemented.

**Important:** Do not execute callback methods explicitly.

***Table 20–1: Callback Methods***

| Callback Name | Implemented for | Method Type | Private Method? |
|---|---|---|---|
| %OnAddToSaveSet() | %RegisteredObject | Instance | Yes |
| %OnAfterBuildIndices() | %Persistent | Class | Yes |
| %OnAfterDelete() | %Persistent | Class | Yes |
| %OnAfterPurgeIndices() | %Persistent | Class | Yes |
| %OnAfterSave() | %Persistent | Instance | Yes |
| %OnBeforeBuildIndices() | %Persistent | Class | Yes |
| %OnBeforePurgeIndices() | %Persistent | Class | Yes |
| %OnBeforeSave() | %Persistent | Instance | Yes |
| %OnClose() | %RegisteredObject | Instance | Yes |
| %OnConstructClone() | %RegisteredObject | Instance | Yes |
| %OnDelete() | %Persistent | Class | Yes |
| %OnNew() | %RegisteredObject | Instance | Yes |
| %OnOpen() | %Persistent, %SerialObject | Instance | Yes |
| %OnReload | %Persistent | Instance | Yes |
| %OnRollBack | %Persistent | Instance | Yes |
| %OnValidateObject() | %RegisteredObject | Instance | Yes |
| %OnDetermineClass() | %CacheStorage | Class | No |

**Note:** For all callbacks that are private methods, documentation for them is only visible in the Class Reference if the **Private** check box in the upper-right corner of the Class Reference is selected.

# 20.1 Callbacks and Triggers

For an application that uses both SQL and object access, if you implement a trigger, it is generally desirable to call the same logic at an equivalent point in object access. For example, if you insert an audit record when a row is deleted, you should probably also insert an audit record if an object is deleted.

If a trigger is defined with `Foreach = row/object`, then the trigger is also called at specific points during object access. See "Triggers and Transactions" in "Using Triggers" in *Using Caché SQL*.

If, however, you cannot create such triggers, and if you want the SQL and object behavior to be synchronized in the sense described previously, then it is necessary to implement one or more callbacks. In these implementations, use logic equivalent to that used in the trigger definitions. Note that the following callback methods have functionality equivalent to that of SQL triggers:

- **%OnBeforeSave()** — BEFORE INSERT, BEFORE UPDATE
- **%OnAfterSave()** — AFTER INSERT, AFTER UPDATE
- **%OnDelete()** — BEFORE DELETE

For more information on triggers, see the "Using Triggers" chapter in *Using Caché SQL* or the CREATE TRIGGER page in the *Caché SQL Reference*.

# 20.2 %OnAddToSaveSet()

This instance method is called whenever the current object is being added to a SaveSet by **%AddToSaveSet()**. **%AddToSaveSet()** can be called by:

- **%Save()** for an instance of %Persistent
- **%GetSwizzleObject()** for an instance of %SerialObject
- **%AddToSaveSet()** for a referencing object

If **%OnAddToSaveSet()** modifies another object, then it is the responsibility of **%OnAddToSaveSet()** to invoke **%AddToSaveSet()** on that modified object. When calling **%AddToSaveSet()** from **%OnAddToSaveSet()**, pass the depth as the first argument and 1 (literal one) as the second argument.

When you invoke **%Save()** on an object, called, for example, *MyPerson*, the system generates a list of objects that *MyPerson* references. A SaveSet is the list of objects consisting of the object to be saved and all the objects that it references. In the example, the SaveSet might include referenced objects *MySpouse*, *MyDoctor*, and so on. For a fuller discussion, see "Saving Objects" in the chapter "Working with Persistent Objects."

The signature of **%OnAddToSaveSet()** is:

### Class Member

```
Method %OnAddToSaveSet(depth As %Integer,
                       insert As %Integer,
                       callcount As %Integer)
                        As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *depth* | An integer value passed in from **%AddToSaveSet()** that represents the internal state of SaveSet construction. If you use **%OnAddToSaveSet()** to add any other objects to the SaveSet, pass this value to **%AddToSaveSet()** without change. |
| *insert* | A flag indicating if the object being saved is being inserted into the extent (1) or that it is already part of the extent (0). |
| *callcount* | The number of times that **%OnAddToSaveSet** has been called for this object. Due to the networked nature of object references, it is possible that **%AddToSaveSet** can be invoked on the same object multiple times. |

The method returns a %Status code, where a failure status causes the save to fail and the transaction to be rolled back.

You can update objects, create new objects, delete objects and ask objects to include themselves in the current SaveSet by calling **%AddToSaveSet()**. If you modify the current instance or any of its descendants, you must let the system know that you have done this; to do so, call **%AddToSaveSet()** for the modified instance(s) and specify the *Refresh* argument as 1.

None of the modification restrictions imposed on **%OnAfterSave()**, **%OnBeforeSave()**, or **%OnValidateObject()** are in place for **%OnAddToSaveSet()**.

If you delete an object using **%OnAddToSaveSet()**, be sure to call **%RemoveFromSaveSet()** to clean up any dangling references to it.

This method can be overridden in any subclass of %Library.RegisteredObject.

# 20.3 %OnAfterBuildIndices()

This class method is called by the **%BuildIndices()** method after that method builds the indices and executes $SortEnd and just before the method releases the extent lock (if one had been requested).

Its signature is:

```
ClassMethod %OnAfterBuildIndices(indexlist As %String(MAXLEN="") = "") As %Status [ Abstract, Private,
 ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *indexlist* | A $List of the index names. |

# 20.4 %OnAfterDelete()

This class method is called by the **%Delete()** method just after a specified object is deleted (immediately after a successful call to **%DeleteData()**). This method allows you to perform actions outside the scope of the object being saved, such as queuing a later notification action.

Its signature is:

### Class Member

```
ClassMethod %OnAfterDelete(oid As %ObjectIdentity) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *oid* | The object being deleted. |

The method returns a %Status code, where a failure status causes **%Delete()** to fail and, if there is an active transaction, to roll it back. If **%Delete()** returns an error (either its own error or one originating in **%DeleteData()**), then there is no call to **%OnAfterDelete()**.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.5 %OnAfterPurgeIndices()

This class method is called by the **%PurgeIndices()** method after that method has completed all its processing.

Its signature is:

```
ClassMethod %OnAfterPurgeIndices(indexlist As %String(MAXLEN="") = "") As %Status [ Abstract, Private,
 ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *indexlist* | A $List of the index names. |

# 20.6 %OnAfterSave()

This instance method is called by the **%Save()** method just after an object is saved. This method allows you to perform actions outside the scope of the object being saved, such as queueing a later notification action. An example is a bank using the deposit in excess of a certain amount to cause it to send the customer an explanation of its deposit policies.

Its signature is:

### Class Member

```
Method %OnAfterSave(insert as %Boolean)As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *insert* | A flag indicating if the object being saved is being inserted into the extent (1) or that it is an update of an existing object (0). |

The method returns a %Status code, where a failure status causes **%Save()** to fail and ultimately roll back the transaction.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.7 %OnBeforeBuildIndices()

This class method is called by the **%BuildIndices()** method after that method acquires the extent lock (if one had been requested) and before that method starts to build indices.

Its signature is:

```
ClassMethod %OnBeforeBuildIndices(ByRef indexlist As %String(MAXLEN="") = "") As %Status [ Abstract,
Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *indexlist* | A $List of the index names. This parameter is passed by reference. If the implementation of **%OnBeforeBuildIndices()** alters this value, then **%BuildIndices()** receives the changed value. |

# 20.8 %OnBeforePurgeIndices()

This class method is called by the **%PurgeIndices()** method before that method starts work. If this method returns an error, then **%PurgeIndices()** will exit immediately without purging any index structures, returning the error to the caller of **%PurgeIndices()**.

Its signature is:

```
ClassMethod %OnBeforePurgeIndices(ByRef indexlist As %String(MAXLEN="") = "") As %Status [ Abstract,
Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *indexlist* | A $List of the index names. This parameter is passed by reference. If the implementation of **%OnBeforePurgeIndices()** alters this value, then **%PurgeIndices()** receives the changed value. |

# 20.9 %OnBeforeSave()

This instance method is called by the **%Save()** method just before an object is saved. This method allows you to request user confirmation before completing an action before saving the instance to disk.

**Important:** It is not valid to modify the current object in **%OnBeforeSave()**. If you wish to modify the object before saving it, implement the **%OnAddToSaveSet()** callback instead and include your logic in that method.

Its signature is:

### Class Member

```
Method %OnBeforeSave(insert as %Boolean) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *insert* | A flag indicating if the object being saved is being inserted into the extent (1) or that it is already part of the extent (0). |

The method returns a %Status code, where a failure status causes the save to fail.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.10 %OnClose()

This instance method is called immediately before an object is destructed, thereby providing the user with an opportunity to perform operations on any ancillary items, such as releasing locks or removing temporary data structures.

Its signature is:

### Class Member

```
Method %OnClose() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

The method returns a %Status code, where a failure status is only informational and does nothing to prevent the object from being destructed.

Subclasses of %Library.RegisteredObject have the option of overriding this method.

# 20.11 %OnConstructClone()

This instance method is called by the **%ConstructClone()** method immediately after the structures have been allocated for the cloned object and all the data has been copied into it. The method allows you to perform any additional actions related to the cloned object, such as taking out a lock or resetting any of the property values of the clone.

Its signature is:

### Class Member

```
Method %OnConstructClone(object As %RegisteredObject,
                         deep As %Boolean,
                         ByRef cloned As %String)
                         As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *object* | The OREF of the object that was cloned. |
| *deep* | How "deep" the cloning process is, where 0 specifies that the clone points to the same related objects as the original; 1 causes objects related to the object being cloned to also be cloned, so that the clone gets its own set of related objects. |
| *cloned* | An argument whose use varies according to how **%OnConstructClone()** is being invoked. See class documentation on %Library.RegisteredObject for details. |

The method returns a %Status code, where a failure status prevents the clone from being created.

Subclasses of %Library.RegisteredObject have the option of overriding this method.

# 20.12 %OnDelete()

This class method is called by the **%Delete()** method just before an object is deleted. This method can be used to ensure that deleting an object does not corrupt data integrity, such as by ensuring that an object designed to contain other objects is only deleted when it is empty.

Its signature is:

### Class Member

```
ClassMethod %OnDelete(oid As %ObjectIdentity) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *oid* | An object identifier for the object being deleted. |

The method returns a %Status code, where a failure status stops the deletion.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.13 %OnNew()

This instance method is called by the **%New()** method at the point when the memory for an object has been allocated and properties are initialized.

Its signature is:

### Class Member

```
Method %OnNew(initvalue As %String) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

where:

| | |
|---|---|
| *initvalue* | A string that the method uses in setting up the object, unless being overridden, as described in the next note. |

**Important:** The arguments for **%OnNew()** must match those of **%New()**. When customizing this method, override the arguments with whatever variables and types that you expect to receive from **%New()**. For example, if **%New()** accepts two arguments — *dob* for a date of birth and *name* for a first name and surname, the signature of **%OnNew()** might be:

### Class Member

```
Method %OnNew(dob as %Date = "", name as %Name = "") as %Status [ Private, ServerOnly =
 1 ]
{
 // body of method here...
}
```

The method returns a %Status code, where a failure status stops the creation of the object.

For example, with a class whose instances must have a value for their Name property, the callback might be of the form:

### Class Member

```
Method %OnNew(initvalue As %String) As %Status
{
    If initvalue="" Quit $$$ERROR($$$GeneralError,"Must supply a name")
    Set ..Name=initvalue
    Quit $$$OK
}
```

Subclasses of %Library.RegisteredObject have the option of overriding this method.

# 20.14 %OnOpen()

This instance method is called by the **%Open()** method just before an object is opened. It allows you to verify the state of an instance compared to any relevant entities.

Its signature is:

### Class Member

```
Method %OnOpen() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

The method returns a %Status code, where a failure status stops the opening of the object.

Subclasses of %Library.Persistent and %SerialObject have the option of overriding this method.

# 20.15 %OnReload

This instance method is called by the **%Reload** method to provide notification that the object specified by *oid* was reloaded. Note that **%Open** calls **%Reload** when the object identified by *oid* is already in memory. If this method returns an error, the object is not opened.

Its signature is:

### Class Member

```
Method %OnReload() As %Status [ Private, ServerOnly = 1 ]
  {
    // body of method here...
  }
```

The method returns a %Status code, where a failure status stops the rollback operation.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.16 %OnRollBack()

Caché calls this instance method when it rolls back an object that it had previously successfully serialized as part of a SaveSet. (See "Saving Objects" in the chapter "Working with Persistent Objects.")

When you invoke **%Save()** for a persistent object or a stream or when you invoke **%GetSwizzleObject()** for a serial object, the system starts a *save transaction* which includes all the objects in the SaveSet. If the **%Save()** fails (because properties do not pass validation, for example), Caché rolls back all objects that it had previously successfully serialized as part of a SaveSet. That is, for each of these objects, Caché invokes **%RollBack()**, which calls **%OnRollBack()**.

Caché does not invoke this method for an object that has not been successfully serialized, that is, an object that is not valid.

**%RollBack()** restores the on-disk state of data for that object to its pre-transaction state, but does not affect the in-memory state of any properties of that object that you have set, apart from its ID assignment. (For more details, see "Saving Objects" in the chapter "Working with Persistent Objects.") If you want to revert in-memory changes, do so in **%OnRollBack()**.

The signature of this method is:

### Class Member

```
Method %OnRollBack() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

The method returns a %Status code, where a failure status stops the rollback operation.

Subclasses of %Library.Persistent have the option of overriding this method.

# 20.17 %OnValidateObject()

This instance method is called by the **%ValidateObject()** method just after all validation has occurred. This allows you to perform custom validation, such as where valid values for one property vary according to the value of another property.

Its signature is:

### Class Member

```
Method %OnValidateObject() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

The method returns a %Status code, where a failure status causes the validation to fail.

Subclasses of %Library.RegisteredObject have the option of overriding this method.

# 20.18 %OnDetermineClass()

The **%OnDetermineClass()** class method returns the most specific type class of that object. (For an introduction to the most specific type class, see "%ClassName() and the Most Specific Type Class (MSTC)" in the chapter "Working with Registered Objects.") **%OnDetermineClass()** is implemented by the default storage class. If you use custom storage or SQL storage, there is no default implementation for this method, but you can implement it.

Its signature is:

```
ClassMethod %OnDetermineClass(
        oid As %ObjectIdentity,
        ByRef class As %String)
    As %Status [ ServerOnly = 1 ]
```

where:

- *oid* is the object identity of an object.
- *class* is the most specific type class of the instance identified by *oid*. The *most specific type class* of an object is the class of which the object is an instance and the object is not an instance of any subclass of that class.

The return value is a status value indicating success or failure.

Subclasses of %Library.SwizzleObject have the option of overriding this method.

## 20.18.1 Invoking %OnDetermineClass()

**%OnDetermineClass()** can be invoked in either of two ways:

```
Set status = ##class(APackage.AClass).%OnDetermineClass(myoid, .myclass)
Set status = myinstance.%OnDetermineClass(myoid, .myclass)
```

where *myoid* is the object whose most specific type class is being determined and *myclass* is the class identified. APackage.AClass is the class from which the method is being invoked and *myinstance* is the instance from which the method is being invoked.

In this case, the method is computing the most specific type class for *myoid* and setting *myclass* equal to that value. If *myoid* is not an instance of the current class, an error is returned.

Consider the example of using **%OnDetermineClass()** with Sample.Employee, which is a subclass of Sample.Person. If there is a call of the form

```
Set status = ##class(Sample.Employee).%OnDetermineClass(myoid, .class)
```

and *myoid* refers to an object whose most specific type class is Sample.Person, then the call returns an error.

## 20.18.2 An Example of Results of Calls to %OnDetermineClass()

Suppose there is a MyPackage.GradStudent class that extends a MyPackage.Student class that extends a MyPackage.Person class. The following shows the results of invoking **%OnDetermineClass()**, passing in the OID of an object whose most specific type class is MyPackage.Student:

- `##class(MyPackage.Person).%OnDetermineClass(myOid,.myClass)`
  - Return value: $$$OK
  - *myClass* set to: MyPackage.Student
- `##class(MyPackage.Student).%OnDetermineClass(myOid,.myClass)`
  - Return value: $$$OK
  - *myClass* set to: MyPackage.Student
- `##class(MyPackage.GradStudent).%OnDetermineClass(myOid,.myClass)`
  - Return value: error status
  - *myClass* set to: `""`

# 21

# Using and Overriding Property Methods

This chapter describes property methods, which are the actual methods that Caché uses when you use OREFs to work with the properties of objects. It discusses the following topics:

- Introduction
- Property accessors for literal properties
- Property accessors for object-valued properties
- How to override a property getter method
- How to override a property setter method
- How to define an object-valued property with a custom accessor method

When viewing this book online, use the preface of this book to quickly find related topics.

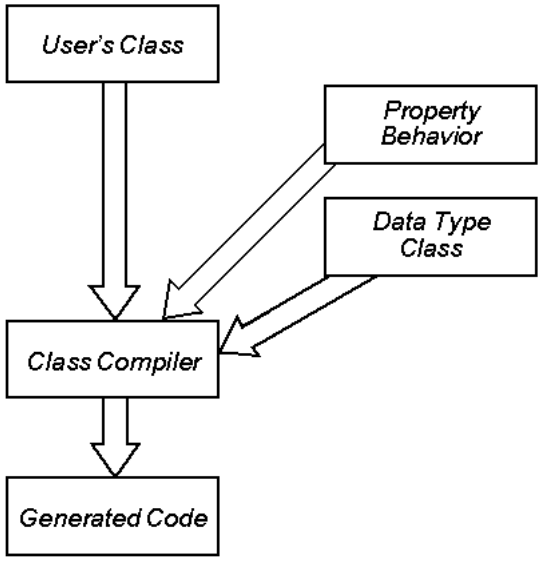
## 21.1 Introduction to Property Methods

Properties have a number of methods associated with them automatically. These methods are not inherited via standard inheritance. Rather, they use a special property behavior mechanism to generate a series of methods for each property.

Each property inherits a set of methods from two places:

- The %Property class, which provides certain built-in behavior, such as **Get()**, **Set()**, and validation code.
- The data type class used by the property, if applicable. Many of these methods are method generators.

*Figure 21–1: Property Behavior*




The property behavior classes are system classes. You cannot specify or modify property behavior.

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 compiled Person class has a set of methods automatically generated for each of its properties. These methods are inherited from the system Property class as well as the data type class associated with the property. The names of these generated methods are 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.

# 21.2 Property Accessors for Literal Properties

The Caché dot syntax for referring to object properties is an interface for a set of accessor methods to retrieve and set values. For each non-calculated property, whenever the code refers to **oref.Prop** (where *oref* is an object and Prop is a property), it is executed as if a system-supplied **PropGet()** or **PropSet()** method were invoked. For example:

### ObjectScript

```
 Set person.DOB = x
```

acts as if the following method was called:

**ObjectScript**

```
Do person.DOBSet(x)
```

while:

**ObjectScript**

```
Write person.Name
```

acts like:

**ObjectScript**

```
Write person.NameGet()
```

In most cases, you cannot see the actual **PropGet()** and **PropSet()** methods; access for simple properties is implemented directly within the Caché virtual machine for optimal performance. You can, however, provide **PropGet()** and **PropSet()** methods for a specific property, as long as that property is not object-typed or multidimensional. If you define these methods, the system automatically invokes them at runtime. The following sections describe how to define these accessor methods. Within the custom methods, you can perform any special processing that your application requires.

Note that the last screen of the **New Property Wizard** in Studio provides options for creating a custom **Get()** method, **Set()**, or both. If you use these options, Studio defines stub methods with suitable signatures.

Accessing the properties of an object by using the **PropGet()** and **PropSet()** methods requires that the object be loaded into memory. On the other hand, the **PropGetStored()** method allows you to retrieve the property value of a stored object directly from disk, without having to load the entire object into memory. For example, to write the name of the person with ID 44, you could use:

**ObjectScript**

```
Write ##class(MyApp.Person).NameGetStored(44)
```

# 21.3 Property Accessors for Object-Valued Properties

For every reference property there are **SetObject()** (using OID value) and **SetObjectId()** (using ID value) methods. For example, to assign a particular saved Person object as the owner of a Car object, use the following code:

**ObjectScript**

```
Do car.OwnerSetObjectId(PersonId)
```

where *car* is the OREF of the Car object and *PersonId* is the ID of the saved Person object.

There are also **GetObject()** and **GetObjectId()** methods, which get the OID or ID associated with the reference property, respectively. Taken all together, the various methods are:

- **GetObject()** — Gets the OID associated with the property. For a property named prop, the method name is **propGetObject()**.
- **GetObjectId()** — Gets the ID associated with the property. For a property named prop, the method name is **propGetObjectId()**.

- **SetObject()** — Sets the OID associated with the property. For a property named prop, the method name is **propSetObject()**.
- **SetObjectId()** — Sets the ID associated with the property. For a property named prop, the method name is **propSetObjectId()**.

# 21.4 Overriding a Property Getter Method

To override the getter method for a property, modify the class that contains the property and add a method as follows:

- It must have the name *PropertyName*`Get`, where *PropertyName* is the name of the corresponding property.
- It takes no arguments.
- Its return type must be the same as the type of the property.
- It must return the value of the property.
- To refer to the value of this property, this method must use the variable `i%`*PropertyName*. This name is case-sensitive.

  **Important:** Within this getter method for a given property, do not use `..`*PropertyName* syntax to refer to the value of that property. If you attempt to do so, the result is a <FRAMESTACK> error, caused by a recursive series of references. You can, however, use `..`*PropertyName* to refer to other properties, because doing so does not cause any recursion.

  The variable `i%`*PropertyName* is an *instance variable*. For more information on instance variables, see "i%PropertyName" in the chapter "Working with Registered Objects."

**Note:** Note that it is not supported to override accessor methods for object-typed properties or for multidimensional properties. Also because the maximum length of a method name is 220 characters, it is not possible to create accessor methods for properties that are 218, 219, or 220 characters long.

The following shows an example, a setter method for a property named HasValue, which is of type %Boolean:

### Class Member

```
Method HasValueGet() As %Boolean
{
  If ((i%NodeType="element")||(i%NodeType="")) Quit 0
  Quit 1
}
```

# 21.5 Overriding a Property Setter Method

To override the setter method for a property, modify the class that contains the property and add a method as follows:

- It must have the name *PropertyName*`Set`, where *PropertyName* is the name of the corresponding property.
- It takes one argument, which contains the value of the property.

  Specifically, this is the value specified in the SET command, when the property is being set.
- It must return a %Status value.
- To set the value of this property, this method must set the variable `i%`*PropertyName*. This name is case-sensitive.

**Important:** Within this setter method for a given property, do not use `..PropertyName` syntax to refer to the value of that property. If you attempt to do so, the result is a <FRAMESTACK> error, caused by a recursive series of references. You can, however, use `..PropertyName` to refer to other properties, because doing so does not cause any recursion.

The variable `i%PropertyName` is an *instance variable*. For more information on instance variables, see "i%PropertyName" in the chapter "Working with Registered Objects."

**Note:** Note that it is not supported to override accessor methods for object-typed properties or for multidimensional properties. Also because the maximum length of a method name is 220 characters, it is not possible to create accessor methods for properties that are 218, 219, or 220 characters long.

For example, suppose that MyProp is of type %String. We could define the following setter method:

### Class Member

```
Method MyPropSet(value as %String) As %Status
{
    if i%MyProp="abc" {
        set i%MyProp="corrected value"
    }
    quit $$$OK
}
```

The following shows another example, a setter method for a property named DefaultXmlns, which is of type %String:

### Class Member

```
Method DefaultXmlnsSet(value As %String) As %Status
{
    set i%DefaultXmlns = value
    If ..Namespaces'="" Set ..Namespaces.DefaultXmlns=value
    quit $$$OK
}
```

Notice that this example refers to the Namespaces property of the same object by using the `..PropertyName` syntax. This usage is not an error, because it does not cause any recursion.

# 21.6 Defining an Object-Valued Property with a Custom Accessor Method

As noted earlier, it is not supported to override accessor methods for object-typed properties. If you need to define a property that holds object values and you need to define custom accessor methods, define the property with the type %CacheString. This is not an object class but is rather a generic class, and it is permitted to override the accessor methods for this property. When using the property, set it equal to an instance of the desired class.

For example, the following class includes the property Zip, whose formal type is %CacheString. The property description indicates that the property is meant to be an instance of Sample.USZipCode. The class also defines the **ZipGet()** and **ZipSet()** property methods.

### Class Definition

```
Class PropMethods.Demo Extends %Persistent
{

/// Timestamp for viewing Zip
Property LastTimeZipViewed As %TimeStamp;
```

```
/// Timestamp for changing Zip
Property LastTimeZipChanged As %TimeStamp;

/// When setting this property, set it equal to instance of Sample.USZipCode.
/// The type is %CacheString rather than Sample.USZipCode, so that it's possible
/// to override ZipGet() and ZipSet().
Property Zip As %CacheString;

Method ZipGet() As %CacheString [ ServerOnly = 1 ]
{
    // get id, swizzle referenced object
    set id = i%Zip
    if (id '= "") {
        set zip = ##class(Sample.USZipCode).%OpenId(id)
        set ..LastTimeZipViewed = $zdt($zts)
    }
    else {
        set zip = ""
    }
    return zip
}

Method ZipSet(zip As %CacheString) As %Status [ ServerOnly = 1 ]
{
    // set i% for new zip
    if ($isobject(zip) && zip.%IsA("Sample.USZipCode")) {
        set id = zip.%Id()
        set i%Zip = id
        set ..LastTimeZipChanged = $zdt($zts)
    }
    else {
        set i%Zip = ""
    }
    quit $$$OK
}

}
```

The following Terminal session demonstrates the use of this class:

```
SAMPLES>set demo=##class(PropMethods.Demo).%New()

SAMPLES>write demo.LastTimeZipChanged

SAMPLES>set zip=##class(Sample.USZipCode).%OpenId(10001)

SAMPLES>set demo.Zip=zip

SAMPLES>w demo.LastTimeZipChanged
10/14/2015 19:21:08
```

# 22

# Defining Data Type Classes

This chapter describes how data type classes work and describes how to define them. It discusses the following topics:

- Overview
- How to define a data type class
- Class methods in data type classes
- Instance methods in data type classes

Also see the chapter "Defining and Using Literal Properties."

When viewing this book online, use the preface of this book to quickly find other topics.

## 22.1 Overview of Data Type Classes

The purpose of a data type class is to be used as the type of a literal property in an object class. Data type classes provide the following features:

- They provide for SQL, ODBC, ActiveX, and Java interoperability by providing SQL logical operation, client data type, and translation information.
- They provide validation for literal data values, which you can extend or customize by using data type class parameters.
- They manage the translation of literal data for its stored (on disk), logical (in memory), and display formats.

See the chapter "Property Methods" for information on how the compiler uses a data type class to generate code for a property.

Data type classes differ from other classes in a number of ways:

- They cannot be instantiated or stored independently.
- They cannot contain properties.
- They support a specific set of methods (called the data type interface), which is described below.

Because it is useful to be aware of some internal details, this section briefly discusses how data type classes work.

As noted previously, the purpose of a data type class is to be used as the type of a property, particularly within a class that extends one of the core object classes. The following shows an example object class that has three properties. Each property uses a data type class as its type.

### Class Definition

```
Class Datatypes.Container Extends %RegisteredObject
{

Property P1 As %String;

Property P2 As %Integer;

Property P3 As %Boolean;

}
```

## 22.1.1 Property Methods

When you add literal properties to a class and compile the class, Caché adds *property methods* to that class. For reference, let us use the term *container class* to refer to the class that contains the properties. The property methods control how the container class handles the data for those properties. This system works as follows:

- Each data type class provides a set of methods, more specifically method generators, that Caché uses when it compiles a class that uses them. A method generator is a method that generates its own runtime code. (For details on method generators, see "Defining Method and Trigger Generators," later in this book.)

  In the example shown here, when we compile the Datatypes.Container, the compiler uses the method generators of the %String, %Integer, and %Boolean data type classes. These method generators create methods for each property and add these methods to the container class. As noted above, these methods are called *property methods*. Their names start with the name of the property to which they apply. For example, for the P1 property, the compiler generates methods such as **P1IsValid()**, **P1Normalize()**, **P1LogicalToDisplay()**, **P1DisplayToLogical()**, and others.

- The container class automatically calls the property methods at suitable points in processing. For example, when you call the **%ValidateObject()** instance method for an instance of the class shown above, the method in turn calls **P1IsValid()**, **P2IsValid()**, and **P3IsValid()** — that is, it calls the **IsValid()** method for each property. For another example, if the container class is persistent, and you use Caché SQL to access all the fields in the associated table, and the SQL runtime mode is ODBC, Caché calls the **LogicalToODBC()** method for each property, so that the query returns results in ODBC format.

Note that the property methods are not visible in the class definition.

## 22.1.2 Data Formats

Many of the property methods translate data from one format to another, for example when displaying data in a human-readable format or when accessing data via ODBC. The formats are:

- *Display* — The format in which data can be input and displayed. For instance, a date in the form of "April 3, 1998" or "23 November, 1977."
- *Logical* — The in-memory format of data, which is the format upon which operations are performed. While dates have the display format described above, their logical format is integer; for the sample dates above, their values in logical format are 57436 and 50000, respectively.
- *Storage* — The on-disk format of data — the format in which data is stored to the database. Typically this is identical to the logical format.
- *ODBC* — The format in which data can be presented via ODBC or JDBC. This format is used when data is exposed to ODBC/SQL. The available formats correspond to those defined by ODBC.
- *XSD* — SOAP-encoded format. This format is used when you export to or import from XML. This applies only to classes that are XML-enabled.

### 22.1.3 Parameters in Data Type Classes

Class parameters have a special behavior when used with data type classes. With data type classes, class parameters are used to provide a way to customize the behavior of any properties based on the data type.

For example, the %Integer data type class has a class parameter, *MAXVAL*, which specifies the maximum valid value for a property of type %Integer. If you define a class with the property NumKids as follows:

#### Class Member

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

This specifies that the *MAXVAL* parameter for the %Integer class will be set to 10 for the NumKids property.

Internally, this works as follows: the validation methods for the standard data type classes are all implemented as method generators and use their various class parameters to control the generation of these validation methods. In this example, this property definition generates content for a **NumKidsIsValid()** method that tests whether the value of NumKids exceeds 10. Without the use of class parameters, creating this functionality would require the definition of an IntegerLessThanTen class.

## 22.2 Defining a Data Type Class

To easily define a data type class, first identify an existing data type class that is close to your needs. Create a subclass of this class. In your subclass:

- Specify suitable values for the keywords SqlCategory, ClientDataType, and OdbcType.
- Override any class parameters as needed. For example, you might override the *MAXLEN* parameter so that there is no length limit for the property.

  If needed, add your own class parameters as well.
- Override the methods of the data type class as needed. In your implementations, refer to the parameters of this class as needed.

If you do not base your data type class on an existing data type class, be sure to add `[ ClassType=datatype ]` to the class definition. This declaration is not needed if you do base your class on another data type class.

**Note:** When defining a data type class, do not extend %Persistent, %RegisteredObject, or other object classes, as data type classes cannot contain properties.

## 22.3 Defining Class Methods in Data Type Classes

Depending on your needs, you should define some or all of the following class methods in your data type classes:

- **IsValid()** — performs validation of data for the property, using property parameters if appropriate. As noted earlier, the **%ValidateObject()** instance method of any object class invokes the **IsValid()** method for each property. This method has the following signature:

  ```
  ClassMethod IsValid(%val) As %Status
  ```

Where *%val* is the value to be validated. This method should return an error status if the value is invalid and should otherwise return $$$OK.

**Note:** It is standard practice in Caché not to invoke validation logic for null values.

- **Normalize()** — converts the data for the property into a standard form or format. The **%NormalizeObject()** instance method of any object class invokes the **Normalize()** method for each property. This method has the following signature:

  ```
  ClassMethod Normalize(%val) As Type
  ```

  Where *%val* is the value to be validated and *Type* is a suitable type class.

- **DisplayToLogical()** — converts a display value into a logical value. (For information on formats, see "Data Formats".) This method has the following signature:

  ```
  ClassMethod DisplayToLogical(%val) As Type
  ```

  Where *%val* is the value to be converted and *Type* is a suitable type class.

  The other format conversion methods have the same general form.

- **LogicalToDisplay()** — converts a logical value to a display value.
- **LogicalToOdbc()** — converts a logical value into an ODBC value.

  Note that the ODBC value must be consistent with the ODBC type specified by the OdbcType class keyword of the data type class.

- **LogicalToStorage()** — converts a logical value into a storage value.
- **LogicalToXSD()** — converts a logical value into the appropriate SOAP-encoded value.
- **OdbcToLogical()** — converts an ODBC value into a logical value.
- **StorageToLogical()** — converts a database storage value into a logical value.
- **XSDToLogical()** — converts a SOAP-encoded value into a logical value.

If the data type class includes the *DISPLAYLIST* and *VALUELIST* parameters, these methods must first check for the presence of these class parameters and include code to process these lists if present. The logic is similar for other methods.

In most cases, many of these methods are method generators. See "Defining Method and Trigger Generators," later in this book.

The following shows an example:

### Class Member

```
ClassMethod LogicalToDate(%val As %MV.Date) As %Library.Date [ CodeMode = expression, ServerOnly = 1
]
{
$s(%val="":"",1:%val+46385)
}
```

**Note:** Note that the data format and translation methods cannot include embedded SQL. If you need to call embedded SQL within this logic, then you can place the embedded SQL in a separate routine, and the method can call this routine.

# 22.4 Defining Instance Methods in Data Type Classes

You can also add instance methods to the data type class, and these methods can use the variable *%val*, which contains the current value of the property. The compiler uses these to generate the associated property methods in any class that uses the data type class. For example, consider the following example data type class:

### Class Definition

```
Class Datatypes.MyDate Extends %Date
{

Method ToMyDate() As %String [ CodeMode = expression ]
{
$ZDate(%val,4)
}
}
```

Suppose that we have another class as follows:

### Class Definition

```
Class Datatypes.Container Extends %Persistent
{

Property DOB As Datatypes.MyDate;

/// additional class members
}
```

When we compile these classes, Caché adds the instance method **DOBToMyDate()** to the container class. Then when we create an instance of the container class, we can invoke this method. For example:

```
SAMPLES>set instance=##class(Datatypes.Container).%New()

SAMPLES>set instance.DOB=+$H

SAMPLES>write instance.DOBToMyDate()
30/10/2014
```

# 23

# Implementing Dynamic Dispatch

This chapter discusses dynamic dispatch in Caché classes. Topics in this chapter include:

- Introduction
- Content of methods that implement dynamic dispatch
- The dynamic dispatch methods

When viewing this book online, use the preface of this book to quickly find other topics.

## 23.1 Introduction to Dynamic Dispatch

Caché classes can include support for what is called *dynamic dispatch*. If dynamic dispatch is in use and a program references a property or method that is not part of the class definition, then a method (called a *dispatch method*) is called that attempts to resolve the undefined method or property. For example, dynamic dispatch can return a value for a property that is not defined or it can invoke a method for a method that is not implemented. The dispatch destination is dynamic in that it does not appear in the class descriptor and is not resolved until runtime.

Caché makes a number of dispatch methods available that you can implement. Each method attempts to resolve an element that is missing under different circumstances.

If you implement a dispatch method, it has the following effects:

- During application execution, if Caché encounters an element that is not part of the compiled class, it invokes the dispatch method to try to resolve the encountered element.
- The application code that uses the class does not do anything special to make this happen. Caché automatically checks for the existence of the dispatch method and, if that method is present, invokes it.

## 23.2 Content of Methods Implementing Dynamic Dispatch

As the application developer, you have control over the content of dispatch methods. The code within them can be whatever is required to implement the methods or properties that the class is attempting to resolve.

Code for dynamic dispatch might include attempts to locate a method based on other classes in the same extent, package, database, on the same file system, or by any other criteria. If a dispatch method provides a general case, it is recommended

that the method also create some kind of log for this action, so that there is a record of any continued operation that includes this general resolution.

For example, the following implementation of **%DispatchClassMethod()** allows the application user to invoke a method to perform whatever action was intended:

### Class Member

```
ClassMethod %DispatchClassMethod(Class As %String, Method As %String, args...)
{
    WRITE "The application has attempted to invoke the following method: ",!,!
    WRITE Class,".",Method,!,!
    WRITE "This method does not exist.",!
    WRITE "Enter the name of the class and method to call",!
    WRITE "or press Enter for both to exit the application.",!,!


    READ "Class name (in the form 'Package.Class'): ",ClassName,!
    READ "Method name: ",MethodName,!

    IF ClassName = "" && MethodName = "" {
        // return a null string to the caller if a return value is expected
        QUIT:$QUIT "" QUIT
    } ELSE {
        // checking $QUIT ensures that a value is returned
        // if and only if it is expected
        IF $QUIT {
            QUIT $CLASSMETHOD(ClassName, MethodName, args...)
        } ELSE {
            DO $CLASSMETHOD(ClassName, MethodName, args...)
            QUIT
        }
    }
}
```

By including this method in a class that is a secondary superclass of all classes in an application, you can establish application-wide handling of calls to nonexistent class methods.

## 23.2.1 Return Values

None of the dispatch methods have specified return values. This is because each should provide output that is of the same type of the call that originally created the need for the dispatch.

If the dispatch method cannot resolve the method or property, it can use **$SYSTEM.Process.ThrowError()** to throw a <METHOD DOES NOT EXIST> or <PROPERTY DOES NOT EXIST> error — or whatever else may be appropriate.

# 23.3 The Dynamic Dispatch Methods

The following methods may be implemented to resolve unknown methods and properties:

- %DispatchMethod()
- %DispatchClassMethod()
- %DispatchGetProperty()
- %DispatchSetProperty()
- %DispatchSetMultidimProperty()

## 23.3.1 %DispatchMethod()

This method implements an unknown method call. Its syntax is:

```
Method %DispatchMethod(Method As %String, Args...)
```

where its first argument is the name of the referenced method and the second argument is an array that holds all the arguments passed to the original method. Since the number of arguments and their types can vary depending on the method being resolved, the code in **%DispatchMethod()** needs to handle them correctly (since the class compiler cannot make any assumptions about the type). The `Args...` syntax handles this flexibly.

Because **%DispatchMethod()** attempts to resolve any unknown instance method associated with the class, it has no specified return value; if successful, it returns a value whose type is determined by the method being resolved and whether the caller expects a return value.

**%DispatchMethod()** can also resolve an unknown multidimensional property reference — that is, to get the value of a property. However, only direct multidimensional property references are supported for dynamic dispatch. **$DATA**, **$ORDER**, and **$QUERY** are not supported, nor is a **SET** command with a list of variables.

## 23.3.2 %DispatchClassMethod()

This method implements an unknown class method call. Its syntax is:

```
ClassMethod %DispatchClassMethod(Class As %String, Method As %String, Args...)
```

where its first two arguments are the name of the referenced class and the name of the referenced method. Its third argument is an array that holds all the arguments passed to the original method. Since the number of arguments and their types can vary depending on the method being resolved, the code in **%DispatchClassMethod()** needs to handle them correctly (since the class compiler cannot make any assumptions about the type). The `Args...` syntax handles this flexibly.

Because **%DispatchClassMethod()** attempts to resolve any unknown class method associated with the class, it has no specified return value; if successful, it returns a value whose type is determined by the method being resolved and whether the caller expects a return value.

## 23.3.3 %DispatchGetProperty()

This method gets the value of an unknown property. Its syntax is:

```
Method %DispatchGetProperty(Property As %String)
```

where its argument is the referenced property. Because **%DispatchGetProperty()** attempts to resolve any unknown property associated with the class, it has no specified return value; if successful, it returns a value whose type is that of the property being resolved

## 23.3.4 %DispatchSetProperty()

This method sets the value of an unknown property. Its syntax is:

```
Method %DispatchSetProperty(Property As %String, Value)
```

where its arguments are the name of the referenced property and the value to set for it.

## 23.3.5 %DispatchSetMultidimProperty()

This method sets the value of an unknown multidimensional property. Its syntax is:

```
Method %DispatchSetMultidimProperty(Property As %String, Value, Subs...)
```

where its first two arguments are the name of the referenced property and the value to set for it. The third argument, *Subs*, is an array that contains the subscript values. *Subs* has an integer value that specifies the number of subscripts, *Subs(1)* has the value of the first subscript, *Subs(2)* has the value of the second, and so on. If no subscripts are given, then *Subs* is undefined.

Only direct multidimensional property references are supported for dynamic dispatch. **$DATA**, **$ORDER**, and **$QUERY** are not supported, nor is a **SET** command with a list of variables.

**Note:** Note that there is no **%DispatchGetMultidimProperty()** dispatch method. This is because a multidimensional property reference is identical to a method call. Thus, such a reference invokes **%DispatchMethod()**, which must include code to differentiate between method names and multidimensional property names.

# A

# Object-Specific ObjectScript Features

ObjectScript includes features specific to working with classes and objects. These are:

- Relative Dot Syntax (..) — For accessing a property or calling a method of the current object.
- ##Class syntax — For invoking a class method, for casting an object reference as another class to call a method, or for accessing the value of a class parameter.
- $this syntax — For getting a handle to the OREF of the current instance, such as for passing it to another class or for another class to refer to properties or methods of the current instance.
- ##super syntax — For invoking a superclass method from within a subclass method.
- Dynamically Accessing Objects — For invoking class methods and instance methods, and for referring to object properties.
- i%<PropertyName> syntax — For referencing an instance variable from within its own **Get** or **Set** accessor method, or bypassing its **Get** or **Set** method.
- ..#<Parameter> syntax — For referencing the value of a class parameter within methods of the same class.

When viewing this book online, use the preface of this book to quickly find related topics.

## A.1 Relative Dot Syntax (..)

The relative dot syntax (..) provides a mechanism for referencing a method or property in the current context. The context for an instance method or a property is the current instance; the context for a class method is the class in which the method is implemented. You cannot use relative dot syntax in a class method to reference properties or instance methods, because these require the instance context.

For example, suppose there is a *Bricks* property of type %Integer:

### Class Member

```
Property Bricks As %Integer;
```

A **CountBricks()** method can then refer to Bricks with relative dot syntax:

### Class Member

```
Method CountBricks()
{
    Write "There are ",..Bricks," bricks.",!
}
```

Similarly, a **WallCheck()** method can refer to **CountBricks()** and *Bricks*:

### Class Member

```
Method WallCheck()
{
    Do ..CountBricks()
    If ..Bricks < 100 {
        Write "Your wall will be small."
    }
}
```

# A.2 ##Class Syntax

The *##class* syntax allows you to:

- Invoke a class method when there is no existing or open instance of a class.
- Cast a method from one class as a method from another.
- Access a class parameter

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

## A.2.1 Invoking a Class Method

To invoke a class method, the syntax is either of the following:

```
>Do ##class(Package.Class).Method(Args)
>Set localname = ##class(Package.Class).Method(Args)
```

It is also valid to use ##class as part of an expression, as in

### ObjectScript

```
 Write ##class(Class).Method(args)*2
```

without setting a variable equal to the return value.

A frequent use of this syntax is in the creation of new instances:

```
>Set LocalInstance = ##class(Package.Class).%New()
```

## A.2.2 Casting a Method

To cast a method of one class as a method of another class, the syntax is either of the following:

```
>Do ##class(Package.Class1)Class2Instance.Method(Args)
>Set localname = ##class(Package.Class1)Class2Instance.Method(Args)
```

You can cast both class methods and instance methods.

For example, suppose that two classes, MyClass.Up and MyClass.Down, both have **Go()** methods. For **MyClass.Up**, this method is as follows

### Class Member

```
Method Go()
{
    Write "Go up.",!
}
```

For **MyClass.Down**, the **Go()** method is as follows:

### Class Member

```
Method Go()
{
    Write "Go down.",!
}
```

You can then create an instance of MyClass.Up and use it to invoke the **MyClass.Down.Go** method:

```
>Set LocalInstance = ##class(MyClass.Up).%New()

>Do ##class(MyClass.Down)LocalInstance.Go()
Go down.
```

It is also valid to use ##class as part of an expression, as in

### ObjectScript

```
 Write ##class(Class).Method(args)*2
```

without setting a variable equal to the return value.

A more generic way to refer to other methods are the $METHOD and $CLASSMETHOD functions, which are for instance and class methods, respectively; these are described in the "Dynamically Accessing Objects" section, later in this chapter. These provide a mechanism for referring to packages, classes, and methods programmatically.

## A.2.3 Accessing a Class Parameter

To access a class parameter, you can use the following expression:

```
##class(Package.Class).#PARMNAME
```

Where *Package.Class* is the name of the class and *PARMNAME* is the name of the parameter. For example:

### ObjectScript

```
 w ##class(%XML.Adaptor).#XMLENABLED
```

displays whether methods generated by the XML adaptor are XML enabled, which by default is set to 1.

You can also use the $PARAMETER functions, which is described in the "Dynamically Accessing Objects" section, later in this chapter.

# A.3 $this Syntax

The *$this* variable provides a handle to the OREF of the current instance, such as for passing it to another class or for another class to refer to the properties or methods of the current instance. When an instance refers to its own properties or methods, relative dot syntax is faster and thus is preferred.

**Note:** *$this* is not case-sensitive; hence, *$this*, *$This*, *$THIS*, or any other variant all have the same value.

For example, suppose there is an application with an Accounting.Order class and an Accounting.Utils class. The **Accounting.Order.CalcTax()** method calls the **Accounting.Utils.GetTaxRate()** and **Accounting.Utils.GetTaxableSubtotal()** methods, passing the city and state values of the current instance to the **GetTaxRate()** method and passing the list of items ordered and relevant tax-related information to **GetTaxableSubtotal()**. **CalcTax()** then uses the values returned to calculate the sales tax for the order. Hence, its code is something like:

### Class Member

```
Method CalcTax() As %Numeric
{
    Set TaxRate = ##Class(Accounting.Utils).GetTaxRate($this)
    Write "The tax rate for ",..City,", ",..State," is ",TaxRate*100,"%",!
    Set TaxableSubtotal = ##class(Accounting.Utils).GetTaxableSubTotal($this)
    Write "The taxable subtotal for this order is $",TaxableSubtotal,!
    Set Tax = TaxableSubtotal * TaxRate
    Write "The tax for this order is $",Tax,!
}
```

The first line of the method uses the *##Class* syntax (described above) to invoke the other method of the class; it passes the current object to that method using the *$this* syntax. The second line of the method uses relative dot syntax to get the values of the *City* and *State* properties. The action on the third line is similar to that on the first line.

In the Accounting.Utils class, the **GetTaxRate()** method can then use the handle to the passed-in instance to get handles to various properties — for both getting and setting their values:

### Class Member

```
ClassMethod GetTaxRate(OrderBeingProcessed As Accounting.Order) As %Numeric
{
    Set LocalCity = OrderBeingProcessed.City
    Set LocalState = OrderBeingProcessed.State
    // code to determine tax rate based on location and set
    // the value of OrderBeingProcessed.TaxRate accordingly
    Quit OrderBeingProcessed.TaxRate
}
```

The **GetTaxableSubtotal()** method also uses the handle to the instance to look at its properties and set the value of its TaxableSubtotal property.

Hence, the output at the Terminal from invoking the **CalcTax()** method for *MyOrder* instance of the Accounting.Order class would be something like:

```
>Do MyOrder.CalcTax()
The tax rate for Cambridge, MA is 5%
The taxable subtotal for this order is $79.82
The tax for this order is $3.99
```

# A.4 ##super Syntax

Suppose that a subclass method overrides a superclass method. From within the subclass method, you can use the *##super()* syntax to invoke the overridden superclass method.

**Note:** *##super* is not case-sensitive. Also note that, unlike other features in this chapter, *##super()* is available within Basic methods as well as within ObjectScript methods.

For example, suppose that the class MyClass.Down extends MyClass.Up and overrides the **Simple** class method. If the code for **MyClass.Up.Simple()** is:

**Class Member**

```
ClassMethod Simple()
{
    Write "Superclass.",!
}
```

and the code for **MyClass.Down.Simple()** is:

**Class Member**

```
ClassMethod Simple()
{
    Write "Subclass.",!
    Do ##super()
}
```

then the output for subclass method, **MyClass.Down.Simple()**, is:

```
>Do ##Class(MyClass.Down).Simple()
Subclass.
Superclass.
>
```

A more generic way to refer to other methods are the $METHOD and $CLASSMETHOD functions, which are for instance and class methods, respectively; these are described in the "Dynamically Accessing Objects" section, later in this chapter. These provide a mechanism for referring to packages, classes, and methods programmatically.

## A.4.1 Calls That ##super Affects

*##super* only affects the current method call. If that method makes any other calls, those calls are relative to the current object or class, *not* the superclass. For example, suppose that MyClass.Up has **MyName()** and **CallMyName()** methods:

```
Class MyClass.Up Extends %Persistent
{

ClassMethod CallMyName()
{
    Do ..MyName()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Up",!
}

}
```

and that MyClass.Down overrides those methods as follows:

```
Class MyClass.Down Extends MyClass.Up
{

ClassMethod CallMyName()
{
    Do ##super()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Down",!
}

}
```

then invoking the **CallMyName()** methods have the following results:

```
USER>d ##class(MyClass.Up).CallMyName()
Called from MyClass.Up

USER>d ##class(MyClass.Down).CallMyName()
Called from MyClass.Down
```

**MyClass.Down.CallMyName()** has different output from **MyClass.Up.CallMyName()** because its **CallMyName()** method includes *##super* and so calls the **MyClass.Up.CallMyName()** method, which then calls the uncast **MyClass.Down.MyName()** method.

## A.4.2 ##super and Method Arguments

*##super* also works with methods that accept arguments. If the subclass method does not specify a default value for an argument, make sure that the method passes the argument by reference to the superclass.

For example, suppose the code for the method in the superclass (**MyClass.Up.SelfAdd()**) is:

### Class Member

```
ClassMethod SelfAdd(Arg As %Integer)
{
    Write Arg,!
    Write Arg + Arg
}
```

then its output is:

```
>Do ##Class(MyClass.Up).SelfAdd(2)
2
4
>
```

The method in the subclass (**MyClass.Down.SelfAdd()**) uses *##super* and passes the argument by reference:

### Class Member

```
ClassMethod SelfAdd(Arg1 As %Integer)
{
    Do ##super(.Arg1)
    Write !
    Write Arg1 + Arg1 + Arg1
}
```

then its output is:

```
>Do ##Class(MyClass.Down).SelfAdd(2)
2
4
6
>
```

In **MyClass.Down.SelfAdd()**, notice the period before the argument name. If we omitted this and we invoked the method without providing an argument, we would receive an <UNDEFINED> error.

# A.5 Dynamically Accessing Objects

Caché supplies several functions that support generalized processing of objects. They do this by allowing a reference to a class and one of its methods or properties to be computed at runtime. (This is known as reflection in Java.) These functions are:

- $CLASSNAME — Returns the name of a class.
- $CLASSMETHOD — Supports calls to a class method.
- $METHOD — Supports calls to an instance method.
- $PARAMETER — Returns the value of a class parameter of the specified class.
- $PROPERTY — Supports references to a particular property of an instance.

The function names are shown here in all uppercase letters, but they are, in fact, not case-sensitive.

## A.5.1 $CLASSNAME

This function returns the name of a class. The signature is:

```
$CLASSNAME(Instance)
```

where *Instance* is an OREF.

For more information, see the $CLASSNAME page in the *Caché ObjectScript Reference*.

## A.5.2 $CLASSMETHOD

This function executes a named class method in the designated class. The signature is:

```
$CLASSMETHOD (Classname, Methodname, Arg1, Arg2, Arg3, ... )
```

where

| | |
|---|---|
| *Classname* | An existing class. |
| *Methodname* | A method of the class specified by the first argument. |
| *Arg1*, *Arg2*, *Arg3*, ... | A series of expressions to be substituted for the arguments to the designated method. |

For more information, see the $CLASSMETHOD page in the *Caché ObjectScript Reference*.

## A.5.3 $METHOD

This function executes a named instance method for a specified instance of a designated class. The signature is:

```
$METHOD (Instance, Methodname, Arg1, Arg2, Arg3, ... )
```

where

| | |
|---|---|
| *Instance* | An OREF of instance of a class. |
| *Methodname* | A method of the class specified by the instance in the first argument. |
| *Arg1*, *Arg2*, *Arg3*, ... | A series of expressions to be substituted for the arguments to the designated method. |

For more information, see the $METHOD page in the *Caché ObjectScript Reference*.

### A.5.4 $PARAMETER

This function returns the value of a class parameter of the designated class. The signature is:

```
$PARAMETER(Instance,Parameter)
```

where:

| | |
|---|---|
| *Instance* | Either the fully qualified name of a class or an OREF of an instance of the class. |
| *Parameter* | A parameter of the given class. |

For more information, see the $PARAMETER page in the *Caché ObjectScript Reference*.

### A.5.5 $PROPERTY

This function gets or sets the value of a property in an instance of the designated class. If the property is multidimensional, the following arguments after the property name are used as indices in accessing the value of the property. The signature is:

```
$PROPERTY (Instance, PropertyName, Index1, Index2, Index3... )
```

where:

| | |
|---|---|
| *Instance* | An OREF of instance of a class. |
| *PropertyName* | A property of the class specified by the instance in the first argument. |
| *Index1*, *Index2*, *Index3*, ... | For multidimensional properties, indices into the array represented by the property. |

For more information, see the $PROPERTY page in the *Caché ObjectScript Reference*.

## A.6 i%<PropertyName> Syntax

This section provides some additional information on instance variables. You do not need to refer to these variables unless you override an *accessor method* for a property; see the chapter "Using and Overriding Property Methods."

When you create an instance of any class, the system creates an instance variable for each non-calculated property of that class. The instance variable holds the value of the property. For the property *PropName*, the instance variable is named *i%PropName*, and this variable name is case-sensitive. These variables are available within any instance method of the class.

For example, if a class has the properties Name and DOB, then the instance variables *i%Name* and *i%DOB* are available within any instance method of the class.

Internally, Caché also uses additional instance variables with names such as *r%PropName* and *m%PropName*, but these are not supported for direct use.

Instance variables have process-private, in-memory storage allocated for them. Note that these variables are not held in the local variable symbol table and are not affected by the **Kill** command.

# A.7 ..#<Parameter> Syntax

The ..#<Parameter> syntax allows for references to class parameters from within methods of the same class.

For example, if a class definition include the following parameter and method:

### Class Member

```
Parameter MyParam = 22;
```

and the following method:

### Class Member

```
ClassMethod WriteMyParam()
{
    Write ..#MyParam
}
```

Then the **WriteMyParam()** method invokes the **Write** command with the value of the *MyParam* parameter as its argument.

# B

# Using the Caché Populate Utility

Caché includes a utility for creating pseudo-random test data for persistent classes. The creation of such data is known as data population; the utility for doing this, known as the Caché *populate utility*, is useful for testing persistent classes before deploying them within a real application. It is especially helpful when testing how various parts of an application will function when working against a large set of data.

The populate utility takes its name from its principal element — the %Populate class, which is part of the Caché class library. Classes that inherit from %Populate contain a method called **Populate()**, which allows you to generate and save class instances containing valid data. You can also customize the behavior of the %Populate class to provide data for your needs.

Along with the %Populate class, the populate utility uses %PopulateUtils. %Populate provides the interface to the utility, while %PopulateUtils is a helper class.

This appendix covers the following topics:

- Basics
- Default behavior
- How to specify the POPSPEC parameter
- How to base one generated property on another
- How %Populate works
- How to implement the OnPopulate() Method (for custom data population)
- Alternative approach to data population

When viewing this book online, use the preface of this book to quickly find related topics.

## B.1 Data Population Basics

To use the populate utility, do the following:

1. Modify each persistent and each serial class that you want to populate with data. Specifically, add %Populate to the end of the list of superclasses, so that the class inherits the interface methods. For example, if a class inherits directly from %Persistent, its new superclass list would be:

   **Class Definition**

   ```
   Class MyApp.MyClass Extends (%Persistent,%Populate) {}
   ```

Do not use %Populate as a *primary* superclass; that is, do not list it as the first class in the superclass list.

Or when using the New Class Wizard within Studio, check **Data Population** on the last screen. This is equivalent to adding the %Populate class to the superclass list.

2. In those classes, optionally specify the *POPSPEC* and *POPORDER* parameters of each property, to control how the populate utility generates data for those properties, if you want to generate custom data rather than the default data, which is described in the next section.

   Later sections of this appendix provide information on these parameters.

3. Recompile the classes.

4. To generate the data, call the **Populate()** method of each persistent class. By default, this method generates 10 records for the class (including any serial objects that it references):

   **ObjectScript**

   ```
   Do ##class(MyApp.MyClass).Populate()
   ```

   If you prefer, you can specify the number of objects to create:

   **ObjectScript**

   ```
   Do ##class(MyApp.MyClass).Populate(num)
   ```

   where *num* is the number of objects that you want.

   Do this in the same order in which you would add records manually for the classes. That is, if Class A has a property that refers to Class B, use the following table to determine which class to populate first:

| If the property in Class A has this form... | And Class B inherits from... | Populate this class first... |
|---|---|---|
| Property PropertyName as ClassB; | %SerialObject | ClassA (this populates ClassB automatically) |
| Property PropertyName as List of ClassB; | | |
| Property PropertyName as Array of ClassB; | | |
| Property PropertyName as ClassB; | %Persistent | ClassB |
| Property PropertyName as List of ClassB; | | |
| Property PropertyName as Array of ClassB; | | |
| Relationship PropertyName as ClassB [ Cardinality = one ...]; | *either* | |
| Relationship PropertyName as ClassB [ Cardinality = parent ...]; | | |
| Relationship PropertyName as ClassB [ Cardinality = many...]; | *either* | ClassA |
| Relationship PropertyName as ClassB [ Cardinality = child ...]; | | |

Later, to remove the generated data, use either the **%DeleteExtent()** method (safe) or the **%KillExtent()** method (fast) of the persistent interface. For more information, see "Deleting Saved Objects" in the chapter "Working with Persistent Objects."

**Tip:** In practice, it is often necessary to populate classes repeatedly, as you make changes to your code. Thus it is useful to write a method or a routine to populate classes in the correct order, as well as to remove the generated data.

### B.1.1 Populate() Details

Formally, the **Populate()** class method has the following signature:

```
classmethod Populate(count As %Integer = 10,
                     verbose As %Integer = 0,
                     DeferIndices As %Integer = 1,
                     ByRef objects As %Integer = 0,
                     tune As %Integer = 1,
                     deterministic As %Integer = 0) as %Integer
```

Where:

- *count* is the desired number of objects to create.
- *verbose* specifies whether the method should print progress messages to the current device.
- *DeferIndices* specifies whether to sort indices after generating the data (true) or while generating the data.
- *objects*, which is passed by reference, is an array that contains the generated objects.
- *tune* specifies whether to run **$SYSTEM.SQL.TuneTable()** after generating the data. If this is 0, the method does not run **$SYSTEM.SQL.TuneTable()**. If this is 1 (the default), the method runs **$SYSTEM.SQL.TuneTable()** for this table. If this is any value higher than 1, the method runs **$SYSTEM.SQL.TuneTable()** for this table and for any tables projected by persistent superclasses of this class.
- *deterministic* specifies whether to generate the same data each time you call the method. By default, the method generates different data each time you call it.

**Populate()** returns the number of objects actually populated:

#### ObjectScript

```
Set objs = ##class(MyApp.MyClass).Populate(100)
// objs is set to the number of objects created.
// objs will be less than or equal to 100
```

In cases with defined constraints, such as a minimum or maximum length, some of the generated data may not pass validation, so that individual objects will not be saved. In these situations, **Populate()** may create fewer than the specified number of objects.

If errors prevent objects from being saved, and this occurs 1000 times sequentially with no successful saves, **Populate()** quits.

## B.2 Default Behavior

This section describes how the **Populate()** method generates data, by default, for the following kinds of properties:

- Literal properties
- Collection properties
- Properties that refer to serial objects
- Properties that refer to persistent objects
- Relationship properties

The **Populate()** method ignores stream properties.

## B.2.1 Literal Properties

This section describes how the **Populate()** method, by default, generates data for properties of the forms:

```
Property PropertyName as Type;
Property PropertyName;
```

Where *Type* is a datatype class.

For these properties, the **Populate()** method first looks at the name. Some property names are handled specially, as follows:

| If the property name is any case variation of the following | Populate() invokes the following method to generate data for it |
|---|---|
| NAME | **Name()** |
| SSN | **SSN()** |
| COMPANY | **Company()** |
| TITLE | **Title()** |
| PHONE | **USPhone()** |
| CITY | **City()** |
| STREET | **Street()** |
| ZIP | **USZip()** |
| MISSION | **Mission()** |
| STATE | **USState()** |
| COLOR | **Color()** |
| PRODUCT | **Product()** |

If the property does not have one of the preceding names, then the **Populate()** method looks at the property type and generates suitable values. For example, if the property type is %String, the **Populate()** method generates random strings (respecting the *MAXLEN* parameter of the property). For another example, if the property type is %Integer, the **Populate()** method generates random integers (respecting the *MINVAL* and *MAXVAL* parameters of the property).

If the property does not have a type, Caché assumes that it is a string. This means that the **Populate()** method generates random strings for its values.

### B.2.1.1 Exceptions

The **Populate()** method does not generate data for a property if the property is private, is multidimensional, is calculated, or has an initial expression.

## B.2.2 Collection Properties

This section describes how the **Populate()** method, by default, generates data for properties of the forms:

```
Property PropertyName as List of Classname;
Property PropertyName as Array of Classname;
```

For such properties:

- If the referenced class is a data type class, the **Populate()** method generates a list or array (as suitable) of values, using the logic described earlier for data type classes.
- If the referenced class is a serial object, the **Populate()** method generates a list or array (as suitable) of serial objects, using the logic described earlier for serial objects.
- If the referenced class is a persistent class, the **Populate()** method performs a random sample of the extent of the referenced class, randomly selects values from that sample, and uses those to generate a list or array (as suitable).

## B.2.3 Properties That Refer to Serial Objects

This section describes how the **Populate()** method, by default, generates data for properties of the form:

```
Property PropertyName as SerialObject;
```

Where *SerialObject* is a class that inherits from %SerialObject.

For such properties:

- If the referenced class inherits from %Populate, the **Populate()** method creates an instance of the class and generates property values as described in the preceding section.
- If the referenced class does not inherit from %Populate, the **Populate()** method does not generate any values for the property.

## B.2.4 Properties That Refer to Persistent Objects

This section describes how the **Populate()** method, by default, generates data for properties of the following form:

```
Property PropertyName as PersistentObject;
```

Where *PersistentObject* is a class that inherits from %Persistent.

For such properties:

- If the referenced class inherits from %Populate, the **Populate()** method performs a random sample of the extent of the referenced class and then randomly selects one value from that sample.

  Note that this means you must generate data for the referenced class first. Or create data for the class in any other way.
- If the referenced class does not inherit from %Populate, the **Populate()** method does not generate any values for the property.

For information on relationships, see the next section.

## B.2.5 Relationship Properties

This section describes how the **Populate()** method, by default, generates data for properties of the following form:

```
Relationship PropertyName as PersistentObject;
```

Where *PersistentObject* is a class that inherits from %Persistent.

For such properties:

- If the referenced class inherits from %Populate:

  - If the cardinality of the relationship is `one` or `parent`, then the **Populate()** method performs a random sample of the extent of the referenced class and then randomly selects one value from that sample.

    Note that this means you must generate data for the referenced class first. Or create data for the class in any other way.

  - If the cardinality of the relationship is `many` or `children`, then the **Populate()** method ignores this property because the values for this property are not stored in the extent for this class.

- If the referenced class does not inherit from %Populate, the **Populate()** method does not generate any values for the property.

# B.3 Specifying the POPSPEC Parameter

For a given property in a class that extends %Populate, you can customize how the **Populate()** method generates data for that property. To do so, do the following:

- Find or create a method that returns a random, but suitable value for this property.

  The %PopulateUtils class provides a large set of such methods; see the Class Reference for details.

- Specify the *POPSPEC* parameter for this property to refer to this method. The first subsection gives the details.

The *POPSPEC* parameter provides additional options for list and array properties, discussed in later subsections.

For a literal, non-collection property, another technique is to identify an SQL table column that contains values to use for this property; then specify the *POPSPEC* parameter to refer to this property; see the last subsection.

**Note:** There is also a *POPSPEC* parameter defined at the class level that controls data population for an entire class. This is an older mechanism (included for compatibility) that is replaced by the property-specific *POPSPEC* parameter. This appendix does not discuss it further.

## B.3.1 Specifying the POPSPEC Parameter for Non-Collection Properties

For a literal property that is not a collection, use one of the following variations:

- `POPSPEC="MethodName()"` — In this case, **Populate()** invokes the class method `MethodName*(` of the %PopulateUtils class.
- `POPSPEC=".MethodName()"` — In this case, **Populate()** invokes the instance method `MethodName()` of the instance that is being generated.
- `POPSPEC="##class(ClassName).MethodName()"` — In this case, **Populate()** invokes the class method `MethodName()` of the `ClassName` class.

For example:

### Class Member

```
Property HomeCity As %String(POPSPEC = "City()");
```

If you need to pass a string value as an argument to the given method, double the starting and closing quotation marks around that string. For example:

#### Class Member

```
Property PName As %String(POPSPEC = "Name(""F"")");
```

Also, you can append a string to the value returned by the specified method. For example:

#### Class Member

```
Property JrName As %String(POPSPEC = "Name()_"" jr."" ");
```

Notice that it is necessary to double the starting and closing quotation marks around that string. It is not possible to prepend a string, because the *POPSPEC* is assumed to start with a method.

Also see "Specifying the POPSPEC Parameter via an SQL Table" for a different approach.

## B.3.2 Specifying the POPSPEC Parameter for List Properties

For a property that is a list of literals or objects, you can use the following variation:

```
POPSPEC="basicspec:MaxNo"
```

Where

- *basicspec* is one of the basic variations shown in the preceding section. Leave *basicspec* empty if the property is a list of objects.
- *MaxNo* is the maximum number of items in the list; the default is 10.

For example:

#### Class Member

```
Property MyListProp As list Of %String(POPSPEC = ".MyInstanceMethod():15");
```

You can omit *basicspec*. For example:

#### Class Member

```
Property Names As list of Name(POPSPEC=":3");
```

In the following examples, there are lists of several types of data. Colors is a list of strings, Kids is a list of references to persistent objects, and Addresses is a list of embedded objects:

```
Property Colors As list of %String(POPSPEC="ValueList("",Red,Green,Blue"")");
Property Kids As list of Person(POPSPEC=":5");
Property Addresses As list of Address(POPSPEC=":3");
```

To generate data for the Colors property, the **Populate()** method calls the **ValueList()** method of the PopulateUtils class. Notice that this example passes a comma-separated list as an argument to this method. For the Kids property, there is no specified method, which results in automatically generated references. For the Addresses property, the serial Address class inherits from %Populate and data is automatically populated for instances of the class.

## B.3.3 Specifying the POPSPEC Parameter for Array Properties

For a property that is an array of literals or objects, you can use the following variation:

```
POPSPEC="basicspec:MaxNo:KeySpecMethod"
```

Where:

- *basicspec* is one of the basic variations shown earlier. Leave *basicspec* empty if the property is a array of objects.
- *MaxNo* is the maximum number of items in the array. The default is 10.
- *KeySpecMethod* is the specification of the method that generates values to use for the keys of the array. The default is `String()`, which means that Caché invokes the **String()** method of %PopulateUtils.

The following examples show arrays of several types of data and different kinds of keys:

```
Property Tix As array of %Integer(POPSPEC="Integer():20:Date()");
Property Reviews As array of Review(POPSPEC=":3:Date()");
Property Actors As array of Actor(POPSPEC=":15:Name()");
```

The Tix property has its data generated using the **Integer()** method of the PopulateUtils class; its keys are generated using the **Date()** method of the PopulateUtils class. The Reviews property has no specified method, which results in automatically generated references, and has its keys also generated using the **Date()** method. The Actors property has no specified method, which results in automatically generated references, and has its keys generated using the **Name()** method of the PopulateUtils class.

## B.3.4 Specifying the POPSPEC Parameter via an SQL Table

For *POPSPEC*, rather than specifying a method that returns a random value, you can specify an SQL table name and an SQL column name to use. If you do so, then the **Populate()** method constructs a dynamic query to return the distinct column values from that column of that table. For this variation of *POPSPEC*, use the following syntax:

```
POPSPEC=":MaxNo:KeySpecMethod:SampleCount:Schema_Table:ColumnName"
```

Where:

- *MaxNo* and *KeySpecMethod* are optional and apply only to collection properties (see earlier the subsections on lists and arrays).
- *SampleCount* is the number of distinct values to retrieve from the given column, to use as a starting point. If this is larger than the number of existing distinct values in that column, then all values are possibly used.
- *Schema_Table* is the name of the table.
- *ColumnName* is the name of the column.

For example:

#### Class Member

```
Property P1 As %String(POPSPEC=":::100:Wasabi_Data.Outlet:Phone");
```

In this example, the property P1 receives a random value from a list of 100 phone numbers retrieved from the Wasabi_Data.Outlet table.

# B.4 Basing One Generated Property on Another

In some cases, the set of suitable value for one property (A) might depend upon the existing value of another property (B). In such a case:

- Create an instance method to generate values for property A. In this method, use instance variables to obtain the value of property B (and any other properties that should be considered). For example:

  **Class Member**

  ```
  Method MyMethod() As %String
  {
      if (i%MyBooleanProperty) {
          quit "abc"
      } else {
          quit "def"
      }
  }
  ```

  For more information on instance variables, see "i%PropertyName" in the chapter "Working with Registered Objects."

  Use this method in the *POPSPEC* parameter of the applicable property. See "Specifying the POPSPEC Parameter", earlier in this appendix.

- Specify the *POPORDER* parameter of any properties that must be populated in a specific order. This parameter should equal an integer. Caché populates properties with lower values of *POPORDER* before properties with higher values of *POPORDER*. For example:

  ```
  Property Name As %String(POPORDER = 2, POPSPEC = ".MyNameMethod()");
  Property Gender As %String(POPORDER = 1, VALUELIST = ",1,2");
  ```

# B.5 How %Populate Works

This section describes how %Populate works internally. The %Populate class contains two method generators: **Populate()** and **PopulateSerial()**. Each persistent or serial class inheriting from %Populate has one or the other of these two methods included in it (as appropriate).

We will describe only the **Populate** method here. The **Populate()** method is a loop, which is repeated for each of the requested number of objects.

Inside the loop, the code:

1. Creates a new object
2. Sets values for its properties
3. Saves and closes the object

A simple property with no overriding *POPSPEC* parameter has a value generated using code with the form:

**ObjectScript**

```
Set obj.Description = ##class(%PopulateUtils).String(50)
```

While using a library method from %PopulateUtils via a "Name:Name()" specification would generate:

**ObjectScript**

```
Set obj.Name = ##class(%PopulateUtils).Name()
```

An embedded Home property might create code like:

### ObjectScript

```
Do obj.HomeSetObject(obj.Home.PopulateSerial())
```

The generator loops through all the properties of the class, and creates code for some of the properties, as follows:

1. It checks if the property is private, is calculated, is multidimensional, or has an initial expression. If any of these are true, the generator exits.
2. If the property is has a *POPSPEC* override, the generator uses that and then exits.
3. If the property is a reference, on the first time through the loop, the generator builds a list of random IDs, takes one from the list, and then exits. For the subsequent passes, the generator simply takes an ID from the list and then exits.
4. If the property name is one of the specially handled names, the generator then uses the corresponding library method and then exits.
5. If the generator can generate code based on the property type, it does so and then exits.
6. Otherwise, the generator sets the property to an empty string.

Refer to the %PopulateUtils class for a list of available methods.

# B.6 Custom Populate Actions and the OnPopulate() Method

For additional control over the generated data, you can define an **OnPopulate()** method. If an **OnPopulate()** method is defined, then the **Populate()** method calls it for each object it generates. The method is called after assigning values to the properties but before the object is saved to disk. Each call to the **Populate()** method results in a check for the existence of the **OnPopulate()** method and a call to **OnPopulate()** it for each object it generates.

This instance method is called by the **Populate** method after assigning values to properties but before the object is saved to disk. This method provides additional control over the generated data. If an **OnPopulate()** method exists, then the **Populate** method calls it for each object that it generates.

Its signature is:

### Class Member

```
Method OnPopulate() As %Status
{
    // body of method here...
}
```

**Note:** This is not a private method.

The method returns a %Status code, where a failure status causes the instance being populated to be discarded.

For example, if you have a stream property, Memo, and wish to assign a value to it when populating, you can provide an **OnPopulate()** method:

### Class Member

```
Method OnPopulate() As %Status
{
    Do ..Memo.Write("Default value")
    QUIT $$$OK
}
```

You can override this method in subclasses of %Library.Populate.

# B.7 Alternative Approach: Creating a Utility Method

There is another way to use the methods of the %Populate and %PopulateUtils classes. Rather than using %Populate as a superclass, write a utility method that generates data for your classes.

In this code, for each class, iterate a desired number of times. In each iteration:

1. Create a new object.
2. Set each property using a suitable random (or nearly random) value.

   To generate data for a property, call a method of %Populate or %PopulateUtils or use your own method.
3. Save the object.

As with the standard approach, it is necessary to generate data for independent classes before generating it for the dependent classes.

For examples of this approach, see the two DeepSee samples in the SAMPLES database, contained in the DeepSee and HoleFoods packages.

## B.7.1 Tips for Building Structure into the Data

In some cases, you might want to include certain values for only a percentage of the cases. You can use the **$RANDOM** function to do this.

In DeepSee.Populate, the sample method **RandomTrue()** returns true or false randomly, depending on a cutoff percentage that you provide as an argument. So, for example, it can return true 10% of the time or 75% of the time. (Internally, this method uses **$RANDOM**.)

When you generate data for a property, you can use this method to determine whether or not to assign a value:

### ObjectScript

```
If ##class(DeepSee.Populate).RandomTrue(15) {
   set object.property="something"
   }
```

In the example shown here, approximately 15 percent of the records will have the given value for this property.

In other cases, you might need to simulate a distribution. To do so, set up and use a lottery system. For example, suppose that 1/4 of the values should be A, 1/4 of the values should be B, and 1/2 the values should be C. The logic for the lottery can go like this:

1. Choose an integer from 1 to 100, inclusive.
2. If the number is less than 25, return value A.
3. If the number is between 25 and 49, inclusive, return value B.
4. Otherwise, return value C.

# C

# Using the %Dictionary Classes

This appendix discusses the *class definition classes*, a set of persistent classes that provide object and SQL access to all class definitions. This appendix discusses the following topics:

- Introduction
- How to browse class definitions
- How to modify class definitions

When viewing this book online, use the preface of this book to quickly find other topics.

## C.1 Introduction to Class Definition Classes

The class definition classes provide object and SQL access to the Caché unified dictionary. Using these classes, you can programmatically examine class definitions, modify class definitions, create new classes, and even write programs that automatically generate documentation. These classes are contained within the %Dictionary package.

**Note:** There is an older set of class definition classes defined within the %Library package. These are maintained for compatibility with existing applications. New code should make use of the classes within the %Dictionary package. Make sure that you specify the correct package name when using these classes or you may inadvertently use the wrong class.

There are two parallel sets of class definition classes: those that represent *defined* classes and those that represent *compiled* classes.

A *defined* class definition represents the definition of a specific class. It includes only information defined by that class; it does not include information inherited from superclasses. In addition to providing information about classes in the dictionary, these classes can be used to programmatically alter or create new class definitions.

A *compiled* class definition includes all of the class members that are inherited from superclasses. A compiled class definition object can only be instantiated from a class that has been compiled. You cannot save a compiled class definition.

This appendix discusses defined class definitions exclusively, though the operation of the compiled class definitions is similar.

The family of class definition classes that represent *defined classes* includes:

| Class | Description |
|---|---|
| %Dictionary.ClassDefinition | Represents a class definition. Contains class keywords as well as collections containing class member definitions. |
| %Dictionary.ForeignKeyDefinition | Represents a foreign key definition within a class. |
| %Dictionary.IndexDefinition | Represents an index definition within a class. |
| %Dictionary.MethodDefinition | Represents a method definition within a class. |
| %Dictionary.ParameterDefinition | Represents a parameter definition within a class. |
| %Dictionary.PropertyDefinition | Represents a property definition within a class. |
| %Dictionary.QueryDefinition | Represents a query definition within a class. |
| %Dictionary.TriggerDefinition | Represents an SQL trigger definition within a class. |

**Important:** To reiterate, the content of an uncompiled class definition (as an instance of the %Dictionary.ClassDefinition) is not necessarily the same as the content of a compiled class definition (as an instance of %Dictionary.CompiledClass). The %Dictionary.ClassDefinition class provides an API to inspect or change the definition of the class — it does not ever represent the compiled class with inheritance resolved; %Dictionary.CompiledClass, on the other hand, *does* represent the compiled class with inheritance resolved.

For example, if you are trying to determine the value of a particular keyword in a class definition, use the *keywordname***IsDefined()** method from %Dictionary.ClassDefinition (such as **OdbcTypeIsDefined()** or **ServerOnlyIsDefined()**). If this boolean method returns false, then the keyword is not explicitly defined for the class. If you check the value of the keyword for the class definition, it will be the default value. However, after compilation (which includes inheritance resolution), the value of the keyword is determined by inheritance and may differ from the value as defined.

# C.2 Browsing Class Definitions

You can use the SQL pages of the Management Portal to browse the class definition classes.

Similarly, you can programmatically browse through the class definitions using the same techniques you would use to browse any other kind of data: you can use the %ResultSet object to iterate over sets of classes and you can instantiate persistent objects that represent specific class definitions.

For example, from within a Caché process, you can get a list of all classes defined within the dictionary for the current namespace by using the **%Dictionary.ClassDefinition:Summary()** query:

### ObjectScript

```
Set result = ##class(%ResultSet).%New("%Dictionary.ClassDefinition:Summary")
Do result.Execute()
While (result.Next()) {
    Write result.Data("Name"),!
}
```

You can just as easily invoke this query from an ActiveX or Java client using the client ResultSet object.

This query will return all of the classes visible from the current namespace (including classes in the system library). You can filter out unwanted classes using the various columns returned by the **%Dictionary.ClassDefinition:Summary()** query.

You can get detailed information about a specific class definition by opening a %Dictionary.ClassDefinition object for the class and observing its properties. The ID used to store %Dictionary.ClassDefinition objects is the class name:

### ObjectScript

```
Set cdef = ##class(%Dictionary.ClassDefinition).%OpenId("Sample.Person")
Write cdef.Name,!

// get list of properties
Set count = cdef.Properties.Count()
For i = 1:1:count {
    Write cdef.Properties.GetAt(i).Name,!
}
```

You can also do this easily from an ActiveX or Java client. Note that you must fully qualify class names with their package name or the call to **%OpenId()** will fail.

# C.3 Altering Class Definitions

You can modify an existing class definition by opening a %Dictionary.ClassDefinition object, making the desired changes, and saving it using the **%Save()** method.

You can create a new class by creating a new %Dictionary.ClassDefinition object, filling in its properties and saving it. When you create %Dictionary.ClassDefinition object, you must pass the name of the class via the **%New()** command. When you want to add a member to the class (such as a property or method), you must create the corresponding definition class (passing its **%New()** command a string containing "class_name.member_name") and add the object to the appropriate collection within the %Dictionary.ClassDefinition object.

For example:

### ObjectScript

```
Set cdef = ##class(%Dictionary.ClassDefinition).%New("MyApp.MyClass")
If $SYSTEM.Status.IsError(cdef)  {
    Do $system.Status.DecomposeStatus(%objlasterror,.Err)
    Write !, Err(Err)
}
Set cdef.Super = "%Persistent,%Populate"

// add a Name property
Set pdef = ##class(%Dictionary.PropertyDefinition).%New("MyClass:Name")
If $SYSTEM.Status.IsError(pdef)  {
    Do $system.Status.DecomposeStatus(%objlasterror,.Err)
    Write !,Err(Err)
}

Do cdef.Properties.Insert(pdef)

Set pdef.Type="%String"

// save the class definition object
Do cdef.%Save()
```

# D

# Using the Object Synchronization Feature

This appendix describes the object synchronization feature, which you can use to synchronize specific tables in databases that are on "occasionally connected" systems. This appendix includes the following sections:

- Introduction
- How to modify the classes to support synchronization
- How to perform the synchronization
- How to translate between GUIDs and OIDs
- How to manually update a SyncTime table

When viewing this book online, use the preface of this book to quickly find other topics.

## D.1 Introduction to Object Synchronization

Object synchronization is a set of tools available with Caché objects that allows application developers to set up a mechanism to synchronize databases on "occasionally connected" systems. By this process, each database updates its objects. Object synchronization offers complementary functionality to Caché system tools that provide high availability and shadowing. Object synchronization is not designed to provide support for real-time updates; rather, it is most useful for a system that needs updates at discrete intervals.

For example, a typical object synchronization application would be in an environment where there is a master copy of a database on a central server and secondary copies on client machines. Consider the case of a sales database, where each sales representative has a copy of the database on a laptop computer. When Mary, a sales representative, is off site, she makes updates to her copy of the database. When she connects her machine to the network, the central and remote copies of the database are synchronized. This can occur hourly, daily, or at any interval.

Object synchronization between two databases involves updating each of them with data from the other. However, Caché does not support bidirectional synchronization as such. Rather, updates from one database are posted to the other; then updates are posted in the opposite direction. For a typical application, if there is a main database and one or more local databases (as in the previous sales database example), it is recommended that updates are from the local to the main database first, and then from the main database to the local one.

For object synchronization, the idea of client and server is by convention only. For any two databases, you can perform bidirectional updates; if there are more than two databases, you can choose what scheme you use to update all of them (such as local databases synchronizing with a main database independently).

This section addresses the following topics:

## D.1.1 The GUID

To ensure that updates work properly, each object in a database should be uniquely distinguishable. To provide this functionality, Caché gives each individual object instance a *GUID* — a globally unique ID. The GUID makes each object universally unique.

The GUID is optionally created, based on the value of the *GUIDENABLED* parameter. If *GUIDENABLED* has a value of 1, then a GUID is assigned to each new object instance.

Consider the following example. Two databases are synchronized and each has the same set of objects in it. After synchronization, each database has a new object added to it. If the two objects share a common GUID, object synchronization considers them the same object in two different states; if each has its own GUID, object synchronization considers them to be different objects.

## D.1.2 How Updates Work

Each update from one database to another is sent as a set of transactions. This ensures that all interdependent objects are updated together. The content of each transaction depends on the contents of the journal for the "source" database. The update can include one or more transactions, up to all transactions that have occurred since the last synchronization.

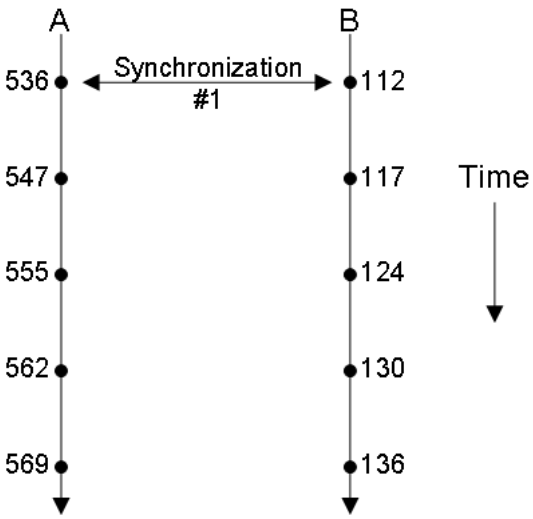
Resolution of the following conditions is the responsibility of the application:

- If two instances that share a unique key have different GUIDs. This requires determining if the two records describe a single object or two unique objects.
- If two changes require reconciliation. This requires determining if the two changes were to a common property or to non-intersecting sets of properties.

## D.1.3 The SyncSet and SyncTime Objects

When two databases are to be synchronized, each has transactions in it that the other lacks. This is illustrated in the following diagram:

*Figure IV–1: Two Unsynchronized Databases*




Here, database A and database B have been synchronized at transaction 536 for database A and transaction 112 for database B. The subsequent transactions for each database need to be updated from each to the other. To do this, Caché uses what is called a SyncSet object. This object contains a list of transactions that are used to update a database. For example, when synchronizing database B with database A, the default contents of the SyncSet object are transactions 547, 555, 562, and 569. Analogously, when synchronizing database A with database B, the default contents of the SyncSet object are transactions 117, 124, 130, and 136. (The transactions do not use a continuous set of numbers, because each transaction encapsulates multiple inserts, updates, and deletes — which themselves use the intermediate numbers.)

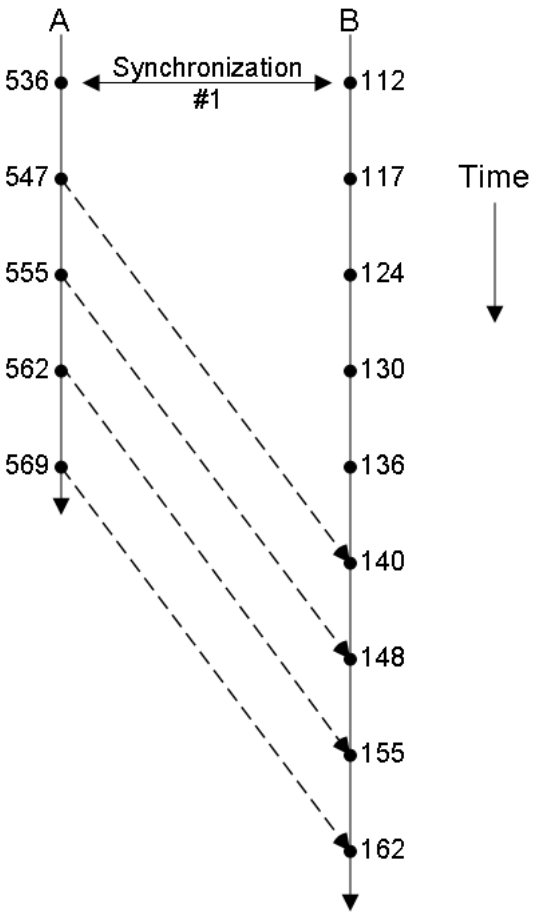
Each database holds a record of its synchronization history with the other. This record is called a SyncTime table. For database, its contents are of the form:

```
Database     Namespace     Last Transaction Sent     Last Transaction Received
-------------------------------------------------------------------------------
B            User          536                       112
```

**Note:** The numbers associated with each transaction do not provide any form of a time stamp. Rather, they indicate the sequence of filing for transactions within an individual database.

Once database B has been synchronized with database A, the two databases might appear as follows:

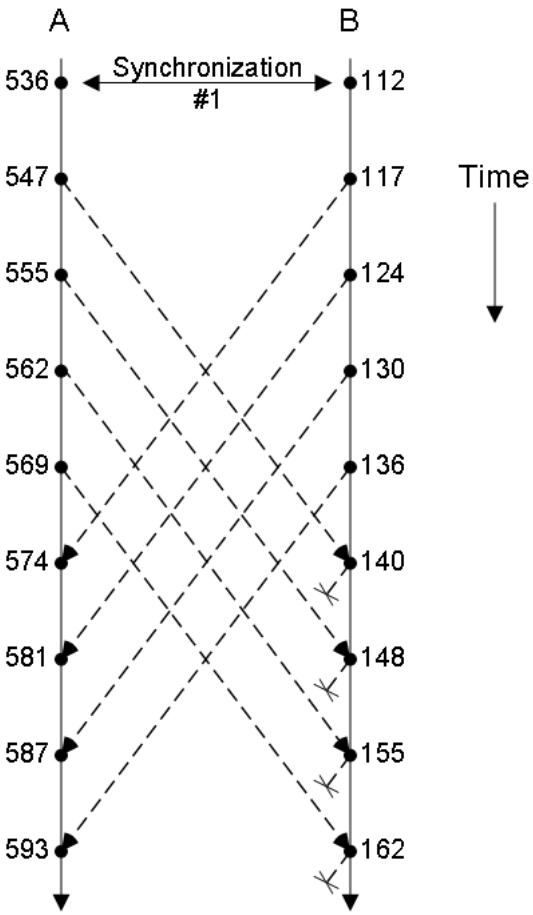
*Figure IV–2: Two Databases, Where One Has Been Synchronized with the Other*




Because the transactions are being added to database B, they result in new transaction numbers in that database.

Analogously, the synchronization of database B with database A results in 117, 124, 130, and 136 being added to database A (and receiving new transaction numbers there):

*Figure IV–3: Two Synchronized Databases*




Note that the transactions from database B that have come from database A (140 through 162) are *not* updated back to database A. This is because the update from B to A uses a special feature that is part of the synchronization functionality. It works as follows:

1. Each transaction in a database is labeled with what can be called "a database of origin." In this example, transaction 140 in database B would be marked as originating in database A, while its transaction 136 would be marked as originating in itself (database B).
2. The **SyncSet.AddTransactions()** method, which bundles a set of transactions for synchronization, allows you to exclude transactions that originate in a particular database. Hence, when updating from B to A, **AddTransactions()** excludes all transactions that originate in database A — because those have already been added to the transaction list for database B.

This functionality prevents creating infinite loops in which two databases continually update each other with the same set of transactions.

# D.2 Modifying the Classes to Support Synchronization

Object synchronization requires that the sites have data with matching sets of GUIDs. If you are starting with an already existing database that does not yet have GUIDs assigned for its records, you need to assign a GUID to each instance (record) in the database, and then make sure there are matching copies of the database on each site. In detail, the process is:

1. For each class being synchronized, set the value of the *OBJJOURNAL* parameter to 1.

   ### Class Member

   ```
   Parameter OBJJOURNAL = 1;
   ```

   This activates the logging of filing operations (that is, insert, update, or delete) within each transaction; this information is stored in the *^OBJ.JournalT* global. An *OBJJOURNAL* value of 1 specifies that the property values that are changed in filing operations are stored in the system journal file; during synchronization, data that needs to be synchronized is retrieved from that file.

   **Note:** *OBJJOURNAL* can also have a value of 2, though the possible use of this value is restricted to special cases. It is *never* for classes using the default storage mechanism (%CacheStorage). A value of 2 specifies that property values that are changed in filing operations are stored in the *^OBJ.Journal* global; during synchronization, data that needs to be synchronized is retrieved from that global. Also, storing information in the global increases the size of the database very quickly.

2. Optionally also set the value of the *JOURNALSTREAM* parameter to 1.

   ### Class Member

   ```
   Parameter JOURNALSTREAM = 1;
   ```

   By default, object synchronization does not support synchronization of file streams. The *JOURNALSTREAM* parameter controls whether or not streams are journaled when *OBJJOURNAL* is true:

   - If *JOURNALSTREAM* is false and *OBJJOURNAL* is true, then objects are journaled but the streams are not.
   - If *JOURNALSTREAM* is true and *OBJJOURNAL* is true, then streams are journaled. Object synchronization tools will process journaled streams when the referencing object is processed.

3. For each class being synchronized, set the value of its *GUIDENABLED* parameter to 1; this tells Caché to allow the class to be stored with GUIDs.

   ### Class Member

   ```
   Parameter GUIDENABLED = 1;
   ```

   Note that if this value is not set, the synchronization does not work properly. Also, you must set *GUIDENABLED* for serial classes, but not for embedded objects.

4. Recompile the class.

5. For each class being synchronized, give each object instance its own GUID by running the **AssignGUID()** method:

   ### ObjectScript

   ```
   Set Status = ##Class(%Library.GUID).AssignGUID(className,displayOutput)
   ```

   where:

   - *className* is the name of class whose instances are receiving GUIDs, such as `"Sample.Person"`.
   - *displayOutput* is an integer where zero specifies that no output is displayed and a nonzero value specifies that output is displayed.

   The method returns a %Status value, which you should check.

6. Put a copy of the database on each site.

# D.3 Performing the Synchronization

This section describes how to perform the synchronization. The database providing the updates is known as the *source database*; and the database receiving the updates is the *target database*. To perform the actual synchronization, the process is:

1. Each time you wish to synchronize the two databases, go to the instance with the source database. On the source database, create a new SyncSet using the **%New()** method of the %SYNC.SyncSet class:

   **ObjectScript**

   ```
   Set SrcSyncSet = ##class(%SYNC.SyncSet).%New("unique_value")
   ```

   The integer argument to **%New()**, *unique_value*, should be an easily identified, unique value. This ensures that each addition to the transaction log on each site can be differentiated from the others.

2. Call the **AddTransactions()** method of the SyncSet instance:

   **ObjectScript**

   ```
   Do SrcSyncSet.AddTransactions(FirstTransaction,LastTransaction,ExcludedDB)
   ```

   Where:

   - *FirstTransaction* is the first transaction number to synchronize.
   - *LastTransaction* is the last transaction number to synchronize.
   - *ExcludedDB* specifies a namespace within a database whose transactions are not included in the SyncSet.

   This method collects the synchronization data and puts it in a global, ready for export.

   Or, to synchronize all transactions since the last synchronization, omit the first and second arguments:

   **ObjectScript**

   ```
   Do SrcSyncSet.AddTransactions(,,ExcludedDB)
   ```

   This gets all transactions, beginning with the first unsynchronized transaction to the most recent transaction. The method uses information in the SyncTime table to determine the values.

   *ExcludedDB* is a $LIST created as follows:

   **ObjectScript**

   ```
   Set ExcludedDB = $ListBuild(GUID,namespace)
   ```

   Where:

   - *GUID* is the system GUID of the target system. This value is available through the %SYS.System.InstanceGUID class method; to invoke this method, use the `##class(%SYS.System).InstanceGUID()` syntax.
   - *namespace* is the namespace on the target system.

3. Call the **ErrCount()** method to determine how many errors were encountered. If there have been errors, the SyncSet.Errors query provides more detailed information.

4. Export the data to a local file using the **ExportFile()** method:

**ObjectScript**

```
Do SrcSyncSet.ExportFile(file,displaymode,bUpdate)
```

Where:

- *file* is the file to which the transactions are being exported; it is a name with a relative or absolute path.
- *displaymode* specifies whether or not the method writes output to the current device. Specify "d" for output or "-d" for no output.
- *bUpdate* is a boolean value that specifies whether or not the SyncTime table is updated (where the default is 1, meaning True). It may be helpful to explicitly set this to 0 at this point, and then set it to 1 after the source receives assurance that the target has indeed received the data and performed the synchronization.

5. Move the exported file from the source machine to the target machine.
6. Create a SyncSet object on the target machine using the **SyncSet.%New()** method. Use the same value for the argument of **%New()** as on the source machine — this is what identifies the source of the synchronized transactions.
7. Read the SyncSet object into the Caché instance on the target machine using the **Import()** method:

**ObjectScript**

```
Set Status = TargetSyncSet.Import(file,lastSync,maxTS,displaymode,errorlog,diag)
```

Where:

- *file* is the file containing the data for import.
- *lastSync* is the last synchronized transaction number (default from synctime table).
- *maxTS* is the last transaction number in the SyncSet object.
- *displaymode* specifies whether or not the method writes output to the current device. Specify "d" for output or "-d" for no output.
- *errorlog* provides a repository for any error information (and is called by reference to provide information for the application).
- *diag* provides more detailed diagnostic information about what is happening when importing

This method puts data into the target database. It behaves as follows:

a. If the method detects that the object has been modified on both the source and target databases since the last synchronization, it invokes the **%ResolveConcurrencyConflict()** callback method; like other callback methods, the content of **%ResolveConcurrencyConflict()** is user-supplied. (Note that this can occur if either the two changes both modified a common property or the two changes each modified non-intersecting sets of properties.) If the **%ResolveConcurrencyConflict()** method is not implemented, then the conflict remains unresolved.

b. If, after the **Import()** method executes, there are unsuccessfully resolved conflicts, these remain in the SyncSet object as unresolved items. Be sure to take the appropriate action regarding the remaining conflicts; this may involve resolution, leaving the items in an unresolved state, and so on.

**Important:** The **Import()** method returns a status value but that status value simply indicates that the method completed without encountering an error that prevented the SyncSet from being processed. It does not indicate that every object in the SyncSet was processed successfully without encountering any errors. For information on synchronization error reporting, see **Import()** in the class reference for %SYNC.SyncSet.

8. Once the first database updates the second database, perform the same process in the other direction so that the second database can update the first one.

# D.4 Translating Between GUIDs and OIDs

To determine the OID of an object from its GUID or vice versa, there are two methods available:

- **%GUIDFind()** is a class method of the %GUID class that takes a GUID of an object instance and returns the OID associated with that instance.
- **%GUID()** is a class method of the %Persistent class that takes an OID of an object instance and returns the GUID associated with that instance; the method can only be run if the *GUIDENABLED* parameter is TRUE for the corresponding class. This method dispatches polymorphically and determines the most specific type class if the OID does not contain that information. If the instance has no GUID, the method returns an empty string.

# D.5 Manually Updating a SyncTime Table

To perform a manual update on the SyncTime table for a database, invoke the **SetlTrn()** method, which sets the last transaction number:

### ObjectScript

```
Set Status=##class(%SYNC.SyncTime).SetlTrn(syncSYSID, syncNSID, ltrn)
```

Where:

- *syncSYSID* is the system GUID of the target system. This value is available through the %SYS.System.InstanceGUID class method; to invoke this method, use the `##class(%SYS.System).InstanceGUID()` syntax.
- *syncNSID* is the namespace on the target system, which is held in the **$Namespace** variable.
- *ltrn* is the highest transaction number known to have been imported. You can get this value by invoking the **GetLastTransaction()** method of the SyncSet.

The **SetlTrn()** method sets the highest transaction number synced in on the target system instead of the default behavior (which is to set the highest transaction number exported from the source system). Either approach is fine and is a choice available during application development.