Orientation Guide for Server-Side Programming

Version 2024.1
2024-07-02

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

Orientation Guide for Server-Side Programming
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

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

1

Introduction to InterSystems IRIS Programming

This page provides a high-level overview of the language elements you can use in InterSystems IRIS® data platform server-side programs.

1.1 Introduction

InterSystems IRIS is a high-performance multi-model data platform with a built-in general-purpose programming language called ObjectScript, as well as built-in support for Python.

InterSystems IRIS supports multiple processes and provides concurrency control. Each process has direct, efficient access to the data.

In InterSystems IRIS, you can write classes, routines, or a mix of these, as suits your preferences. In all cases, stored data is ultimately contained in structures known as globals. InterSystems IRIS programming has the following features:

- Classes and routines can be used interchangeably.
- Classes and routines can call each other.
- Classes provide object-oriented features.
- Database storage is integrated into both ObjectScript and Python.
- Classes can persist data in a way that simplifies programming. If you use persistent classes, data is simultaneously available as objects, SQL tables, and globals.
- You can access globals directly from either classes or routines, which means that you have the flexibility to store and access data exactly how you want.

You can choose the approach that is appropriate for your needs.

1.2 Classes

InterSystems IRIS supports classes. You can use the system classes and you can define your own classes.

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

InterSystems IRIS class definitions use Class Definition Language (CDL) to specify the class and its members such as properties, methods, and parameters. You can use either Python or ObjectScript to write the executable code inside of methods. For each method, specify which language you will be writing the method in by using the Language keyword, as in the example below.

The following shows a class definition:

Class/ObjectScript

```
Class Sample.Employee Extends %Persistent
{

/// The employee's name.
Property Name As %String(MAXLEN = 50);

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

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

/// This method prints employee information using ObjectScript.
Method PrintEmployee() [ Language = objectscript]
{
    Write !,"Name: ", ..Name, " Title: ", ..Title
}

}
```

Class/Python

```
Class Sample.Employee Extends %Persistent
{

/// The employee's name.
Property Name As %String(MAXLEN = 50);

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

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

/// This method prints employee information using Embedded Python.
Method PrintEmployee() [ Language = python ]
{
    print("\nName:", self.Name, "Title:", self.Title)
}

}
```

If you do not specify which language a method uses, the compiler will assume that the method is written in ObjectScript.

Other articles discuss classes in InterSystems IRIS and the unique capabilities of persistent classes in InterSystems IRIS.

1.3 Routines

When you create routines in InterSystems IRIS, you use ObjectScript. The following shows part of a routine written in ObjectScript:

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

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

1.4 Using Classes and Routines Together

In InterSystems IRIS, you can use classes from within routines. For example, the following shows part of a routine, in which we refer to the `Sample.Employee` class:

ObjectScript

```
 //get details of random employee and print them
showemployee() public {
    set rand=$RANDOM(10)+1        ; rand is an integer in the range 1-10
    write "Your random number: "_rand
    set employee=##class(Sample.Employee).%OpenId(rand)
    do employee.PrintEmployee()
    write !,"This employee's salary: "_employee.Salary

    }
```

Similarly, a method can invoke a label in a routine. For example, the following invokes the label `ComputeRaise` in the routine `employeeutils`:

Method/ObjectScript

```
Method RaiseSalary() As %Numeric
{
    set newsalary=$$ComputeRaise^employeeutils(..Salary)
    return newsalary
}
```

Method/Python

```
Method RaiseSalary() as %Numeric [ Language = python ]
{
    import iris
    newsalary=iris.routine("ComputeRaise^employeeutils", self.Salary)
    return newsalary
}
```

1.5 Introduction to Globals

InterSystems IRIS supports a special kind of variable that is not seen in other programming languages; this is a *global variable*, which is usually just called a *global*. In InterSystems IRIS, the term *global* indicates that this data is available to all processes accessing this database. This usage is different from other programming languages in which *global* means "available to all code in this module." The contents of a global are stored in an InterSystems IRIS database.

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

When you use persistent classes, you can create, modify, and delete stored data in the following ways:

- In ObjectScript, using methods such as **%New()**, **%Save()**, **%Open()**, and **%Delete()**.
- In Python, using methods such as **_New()**, **_Save()**, **_Open()**, and **_Delete()**.
- In ObjectScript, using direct global access.
- In Python, using the **gref()** method to provide direct global access.
- By using InterSystems SQL.

Internally, the system always uses direct global access.

Programmers do not necessarily have to work directly with globals, but it can be helpful to know about them and the ways they can be used; see Introduction to Globals.

1.6 InterSystems SQL

InterSystems IRIS provides an implementation of SQL, known as InterSystems SQL. You can use InterSystems SQL within methods and within routines.

1.6.1 Using SQL from ObjectScript

You can execute SQL from ObjectScript using either or both of the following ways:

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

  ObjectScript

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

  You can use dynamic SQL in ObjectScript methods and routines.

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

  ObjectScript

  ```
  &sql(SELECT COUNT(*) INTO :myvar FROM Sample.Employee)
     IF SQLCODE<0 {WRITE "SQLCODE error ",SQLCODE," ",%msg  QUIT}
     ELSEIF SQLCODE=100 {WRITE "Query returns no results"  QUIT}
  WRITE myvar
  ```

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

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

You can use embedded SQL in ObjectScript methods and routines.

1.6.2 Using SQL from Python

Using SQL from Python is similar to using Dynamic SQL from ObjectScript. You can execute SQL from Python using either or both of the following ways:

- You can execute the SQL query directly, as in the following example:

  Python

  ```
  import iris
  rset = iris.sql.exec("SELECT * FROM Sample.Employee ORDER BY Salary")
  for row in rset:
      print(row)
  ```

  The second line executes an InterSystems SQL query and returns a result set stored in the variable `rset`.

- You can also prepare the SQL query first, then execute it, as in the following example:

  Python

  ```
  import iris
  statement = iris.sql.prepare("SELECT * FROM Sample.Employee ORDER BY Salary")
  rset = statement.execute()
  for row in rset:
      print(row)
  ```

  In this example, the second line returns a SQL query which is executed on the third line to return a result set.

You can use either of these approaches to execute SQL queries in the Python terminal or in Python methods.

1.7 Macros

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

ObjectScript

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

For more information on using macros, see Using Macros and Include Files.

1.8 Include Files

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

For more information on how to use include files, see Using Macros and Include Files.

1.9 How These Code Elements Work Together

It is useful to understand how InterSystems IRIS uses the code elements introduced in this page.

The reason that you can use a mix of ObjectScript, Python, InterSystems SQL, class definitions, macros, routines, and so on is that InterSystems IRIS does not *directly* use the code that you write. Instead, when you compile your code, the system generates lower-level code that it uses. This is OBJ code for ObjectScript, used by the ObjectScript engine, and PYC code for Python, used by the Python engine.

There are multiple steps. It is not necessary to know the steps in detail, but the following points are good to remember:

- The *class compiler* processes class definitions and ObjectScript code into INT code for all elements other than Python methods. Python code is processed into PY code.

  In some cases, the compiler generates and saves additional classes, which you should not edit. This occurs, for example, when you compile classes that define web services and web clients.

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

- For ObjectScript code, a *preprocessor* (sometimes called the *macro preprocessor* or *MPP*) uses the include files and replaces the macros. It also handles the embedded SQL in routines.

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

- Additional compilers create INT code for routines.

- INT code and PY code are an intermediate layer in which access to data is handled via direct global access. This code is human-readable.

- INT code is used to generate OBJ code, and PY code is used to generate PYC code. The InterSystems IRIS virtual machine uses this code. Once you have compiled your code into OBJ and PYC code, the INT and PY routines are no longer necessary for code execution.

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

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

Note: All class definitions and routines are stored in the same InterSystems IRIS databases as the generated code. This fact makes the code easier to manage. InterSystems IRIS provides a robust set of source control hooks that InterSystems developers have used for many years. You can use these hooks as well.

2

A Closer Look at ObjectScript

This page gives an overview of the ObjectScript language and provides useful background information for those who want to program in ObjectScript or need to understand code that other people have written.

Both methods and routines can be written in ObjectScript, but most modern code is written using methods. Methods are contained within classes, which allow you to group like methods together, automatically generate documentation in the Class Reference, as well as use the object-oriented capabilities of InterSystems IRIS.

This does not mean routines are not important. Many useful system utilities are written as routines, and routines are generated when you compile a class.

2.1 A Sample Class

The following shows an sample class named `User.DemoClass` that contains methods written in ObjectScript. This example gives us an opportunity to see some common ObjectScript commands, operators, and functions, and to see how code is organized within a method.

Class Definition

```
Class User.DemoClass
{

/// Generate a random number.
/// This method can be called from outside the class.
ClassMethod Random() [ Language = objectscript ]
{
    set rand=$RANDOM(10)+1        ; rand is an integer in the range 1-10
    write "Your random number: "_rand
    set name=..GetNumberName(rand)
    write !, "Name of this number: "_name
}

/// Input a number.
/// This method can be called from outside the class.
ClassMethod Input() [ Language = objectscript ]
{
    read "Enter a number from 1 to 10: ", input
    set name=..GetNumberName(input)
    write !, "Name of this number: "_name
}

/// Given an number, return the name.
/// This method can be called only from within this class.
ClassMethod GetNumberName(number As %Integer) As %Integer [ Language = objectscript, Private ]
{
    set name=$CASE(number,1:"one",2:"two",3:"three",
        4:"four",5:"five",6:"six",7:"seven",8:"eight",
        9:"nine",10:"ten",:"other")
    quit name
}
```

```
/// Write some interesting values.
/// This method can be called from outside the class.
ClassMethod Interesting() [ Language = objectscript ]
{
    write "Today's date: "_$ZDATE($HOROLOG,3)
    write !,"Your installed version: "_$ZVERSION
    write !,"Your username: "_$USERNAME
    write !,"Your security roles: "_$ROLES
}

}
```

Note the following highlights:

- The **Random()** and **Input()** methods invoke the **GetNumberName()** method, which is private to this class and cannot be called from outside the class.
- **WRITE**, **QUIT**, **SET**, and **READ** are commands. The language includes other commands to remove variables, commands to control program flow, commands to control I/O devices, commands to manage transactions (possibly nested), and so on.

  The names of commands are not case-sensitive, although they are shown in running text in all upper case by convention.
- The sample includes two operators. The plus sign (+) performs addition, and the underscore (_) performs string concatenation.

  ObjectScript provides the usual operators and some special operators not seen in other languages.
- **$RANDOM**, **$CASE**, and **$ZDATE** are functions.

  The language provides functions for string operations, conversions of many kinds, formatting operations, mathematical operations, and others.
- **$HOROLOG**, **$ZVERSION**, **$USERNAME**, and **$ROLES** are system variables (called *special variables* in InterSystems IRIS). Most special variables contain values for aspects of the InterSystems IRIS operating environment, the current processing state, and so on.
- ObjectScript supports comment lines, block comments, and comments at the end of statements.

We can execute the methods of this class in the Terminal, as a demonstration. In these examples, `TESTNAMESPACE>` is the prompt shown in the Terminal. The text after the prompt on the same line is the entered command. The lines after that show the values that the system writes to the Terminal in response.

```
TESTNAMESPACE>do ##class(User.DemoClass).Input()
Enter a number from 1 to 10: 7
Name of this number: seven
TESTNAMESPACE>do ##class(User.DemoClass).Interesting()
Today's date: 2021-07-15
Your installed version: IRIS for Windows (x86-64) 2019.3 (Build 310U) Mon Oct 21 2019 13:48:58 EDT
Your username: SuperUser
Your security roles: %All
TESTNAMESPACE>
```

2.2 A Sample Routine

The following shows an sample routine named `demoroutine` that is written in ObjectScript. It contains procedures that do the exact same thing as the methods shown in the sample class in the previous section.

ObjectScript

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

```
 write !,"interesting"
 quit

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

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

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

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

Note the following highlights:

- The only identifiers that actually start with a caret (^) are the names of globals; these are discussed later in this page. However, in running text and in code comments, it is customary to refer to a routine as if its name started with a caret, because you use the caret when you invoke the routine (as shown later in this page). For example, the routine `demoroutine` is usually called `^demoroutine`.
- The routine name does not have to be included within the routine. However, many programmers include the routine name as a comment at the start of the routine or as the first label in the routine.
- The routine has multiple *labels*: `random`, `input`, `getnumbername`, and `interesting`.

  Labels are used to indicate the starting point for procedures (as in this example), functions, and subroutines. You can also use them as a destination for certain commands.

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

  Labels are also called *entry points* or *tags*.
- The `random` and `input` subroutines invoke the `getnumbername` subroutine, which is private to the routine.

We can execute parts of this routine in the Terminal, as a demonstration. First, the following shows a Terminal session, in which we run the routine itself.

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

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

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

```
TESTNAMESPACE>do input^demoroutine
Enter a number from 1 to 10: 7
Name of this number: seven
TESTNAMESPACE>do interesting^demoroutine
Today's date: 2018-02-06
Your installed version: IRIS for Windows (x86-64) 2018.1 (Build 513U) Fri Jan 26 2018 18:35:11 EST
Your username: _SYSTEM
Your security roles: %All
TESTNAMESPACE>
```

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

2.3 Variables

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

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

  Local variables can have public or private scope.

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

There are special kinds of globals not discussed here; see Caret (^) in the article What's That?.

2.3.1 Variable Names

The names of variables follow these rules:

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

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

When you define percent variables, use the following rules:

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

For details on percent variables and variable scope, see Variable Availability and Scope; also see Callable User-defined Code Modules).

For further details on names and for variations, see Syntax Rules. Or see Rules and Guidelines for Identifiers.

2.3.2 Variable Types

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

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

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

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

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

When you create classes, you can specify types for properties, for arguments to methods, and so on. The InterSystems IRIS class mechanisms enforce these types as you would expect.

2.3.3 Variable Length

The length of a value of a variable must be less than the string length limit.

2.3.4 Variable Existence

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

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

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

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

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

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

TESTNAMESPACE>write $DATA(x)
1
```

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

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

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

```
TESTNAMESPACE>WRITE testvar

WRITE testvar
^
<UNDEFINED> *testvar
```

2.4 Variable Availability and Scope

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

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

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

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

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

2.4.1 Summary of Variable Scope

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

- An InterSystems IRIS instance includes multiple namespaces, including multiple system namespaces and probably multiple namespaces that you define.

  A namespace is the environment in which any code runs. Namespaces are discussed later in more detail.
- You can run multiple processes simultaneously in a namespace. In a typical application, many processes are running at the same time.

The following table summarizes where variables are available:

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

*By default, variables defined in a method are private to the method, as noted before. Also, in a method, you can declare variables as public variables, although this practice is not preferred. See PublicList.

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

2.5 Multidimensional Arrays

In ObjectScript, any variable can be an InterSystems IRIS *multidimensional array* (also called an *array*). A multidimensional array is generally intended to hold a set of values that are related in some way. ObjectScript provides commands and functions that provide convenient and fast access to the values.

You may or may not work directly with multidimensional arrays, depending on the system classes that you use and your own preferences. InterSystems IRIS provides a class-based alternative to use when you want a container for sets of related values; see Collection Classes.

2.5.1 Basics

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

ObjectScript

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

This example shows a typical array. Notes:

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

ObjectScript

```
kill myarray
```

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

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

ObjectScript

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

- There is a limit to the possible length of a global reference. This limit affects the length of the global name and the length and number of any subscripts. If you exceed the limit, you get a <SUBSCRIPT> error. See Maximum Length of a Global Reference.
- The length of a value of a node must be less than the string length limit.

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

2.5.2 Structure Variations

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

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

ObjectScript

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

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

ObjectScript

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

2.5.3 Use Notes

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

Kind of Variable	Example and Notes
Local variable without subscripts	`Set MyVar=10` Variables like this are quite common. The majority of the variables you see might be like this.

Local variable with subscripts	`Set MyVar(1)="alpha"`
	`Set MyVar(2)="beta"`
	`Set MyVar(3)="gamma"`
	A local array like this is useful when you want to pass a set of related values.

Global variable without subscripts	`Set ^MyVar="saved note"` In practice, globals usually have subscripts.
Global variable with subscripts	`Set ^MyVar($USERNAME,"Preference 1")=42`

2.6 Passing Variables by Value or by Reference

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

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

When you pass a local variable with no subscripts to 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. To pass the memory address instead, place a period immediately before the name of the variable in the argument list.

To demonstrate this, consider the following method in a class called **Test.Parameters**:

Class Member

```
ClassMethod Square(input As %Integer) As %Integer
{
    set answer=input*input
    set input=input + 10
    return answer
}
```

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

```
TESTNAMESPACE>set myVariable = 5

TESTNAMESPACE>write ##class(Test.Parameters).Square(myVariable)
25
TESTNAMESPACE>write myVariable
5
```

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

```
TESTNAMESPACE>set myVariable = 5

TESTNAMESPACE>write ##class(Test.Parameters).Square(.myVariable)
25
TESTNAMESPACE>write myVariable
15
```

Consider the following method, which writes the contents of the argument it receives:

```
ClassMethod WriteContents(input As %String)
{
    zwrite input
}
```

Now, suppose you have an array with three nodes:

```
TESTNAMESPACE>zwrite myArray
myArray="Hello"
myArray(1)="My"
myArray(2)="Friend"
```

If you pass the array to the method by value, you are only passing the top-level node:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(myArray)
input="Hello"
```

If you pass the array to the method by reference, you are passing the entire array:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(.myArray)
input="Hello"
input(1)="My"
input(2)="Friend"
```

You can pass the value of a single node of a global to a method:

```
TESTNAMESPACE>zwrite ^myGlobal
^myGlobal="Start"
^myGlobal(1)="Your"
^myGlobal(2)="Engines"
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(^myGlobal)
input="Start"
```

Trying to pass a global to a method by reference results in a syntax error:

```
TESTNAMESPACE>do ##class(Test.Parameters).WriteContents(.^myGlobal)
^
<SYNTAX>
```

The following table summarizes all the possibilities:

Kind of Variable	Passing by Value	Passing by Reference
Local variable with no subscripts	The standard way in which these variables are passed	Allowed
Local with subscripts (array)	Passes the value of a single node	The standard way in which these variables are passed
Global variable with or without subscripts	Passes the value of a single node	Cannot be passed this way (data for a global is not in memory)
Object Reference (OREF)*	The standard way in which these variables are passed	Allowed

* If you have a variable representing an object, you refer to the object by means of an object reference (OREF). When you pass an OREF as an argument, you typically pass it by value. However, since an OREF is a pointer to the object, you are effectively passing the object by reference. Changing the value of a property of the object inside the method changes the actual object, not a copy of the object. Passing an OREF by reference is allowed and can be used if you want to change the OREF to point to a different object. This is not a common usage. See Objects for more information on objects and object references.

2.7 Operators

This section provides an overview of the operators in ObjectScript; some are familiar, and others are not.

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

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

2.7.1 Familiar Operators

ObjectScript provides the following operators for common activities:

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

  You can use this immediately before any logical value as well as immediately before a logical comparison operator.
- Operators to combine logical values: AND (&&), OR (||)

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

2.7.2 Unfamiliar Operators

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

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

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

  ObjectScript

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

  This is a valuable tool for ensuring the validity of input data, and you can use it within the definition of class properties.
- The binary contains operator ([) returns 1 (true) or 0 (false) depending on whether the sequence of characters in the right operand is a substring of the left operand. For example:

  ObjectScript

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

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

2.8 Commands

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

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

2.8.1 Familiar Commands

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

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

  There are other commands for flow control, but they are used less often.
- To trap errors, use **TRY** and **CATCH**, which work together.
- To write a value, use **WRITE**. This writes values to the current device (for example, the Terminal or a file).

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

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

  ObjectScript

  ```
  write "hello world",!,"another line"
  ```
- To read a value from the current device (for example, the Terminal), use **READ**.
- To work with devices other than the principal device, use the following commands:
  - **OPEN** makes a device available for use.
  - **USE** specifies an open device as the current device for use by **WRITE** and **READ**.

  - **CLOSE** makes a device no longer available for use.

- To control concurrency, use **LOCK**. Note that the InterSystems IRIS lock management system is different from analogous systems in other languages. It is important to review how it works; see Locking and Concurrency Control.

  You use this command in cases where multiple processes can potentially access the same variable or other item.
- To manage transactions, use **TSTART**, **TCOMMIT**, **TROLLBACK**, and related commands.
- For debugging, use **ZBREAK** and related commands.
- To suspend execution, use **HANG**.

2.8.2 Commands for Use with Multidimensional Arrays

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

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

  For example, the following removes an entire multidimensional array:

  ObjectScript

  ```
  kill myarray
  ```

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

  ObjectScript

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

- To delete a global or a global node but none of its descendent subnodes, use **ZKILL**.
- To iterate through all nodes of a multidimensional array and write them all, use **ZWRITE**. This is particularly convenient in the Terminal. The following sample Terminal session shows what the output looks like:

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

  This example uses a global variable rather than a local one, but remember that both can be multidimensional arrays.
- To copy a set of nodes from one multidimensional array into another, preserving existing nodes in the target if possible, use **MERGE**. For example, the following command copies an entire in-memory array (`sourcearray`) into a new global (`^mytestglobal`):

  ObjectScript

  ```
  MERGE ^mytestglobal=sourcearray
  ```

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

2.9 Special Variables

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

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

- **$HOROLOG**, which contains the date and time for the current process, as given by the operating system. See Date and Time Values.
- **$USERNAME** and **$ROLES**, which contain information about the username currently in use, as well as the roles to which that user belongs.

  ObjectScript

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

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

Others include **$JOB**, **$ZTIMEZONE**, **$IO**, and **$ZDEVICE**.

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

2.9.1 $SYSTEM Special Variable

The special variable **$SYSTEM** provides easy access to a large set of utility methods.

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

ObjectScript

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

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

ObjectScript

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

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

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

Methods of the class: %SYSTEM.OBJ

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

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

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

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

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

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

2.10 Locking and Concurrency Control

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

2.10.1 Basics

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

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

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

The system automatically uses the **LOCK** command internally in many cases, such as when you work with persistent objects or when you use certain InterSystems SQL commands.

2.10.2 The Lock Table

InterSystems IRIS maintains a system-wide, in-memory table that records all current locks and the processes that own them. This table — the lock table — is accessible via the Management Portal, where you can view the locks and (in rare cases, if needed) remove them.

The lock table cannot exceed a fixed size, which you can specify. For information, see Monitoring Locks in the *Monitoring Guide*. Consequently, it is possible for the lock table to fill up, such that no further locks are possible. Filling the lock table is *not* generally considered to be an application error; InterSystems IRIS also provides a lock queue, and processes wait until there is space to add their locks to the lock table.

However, if two processes each assert an incremental lock on a variable already locked by the other process, that is a condition called *deadlock* and it *is* considered an application programming error. For details, see Avoiding Deadlock in *Using ObjectScript*.

2.11 System Functions

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

The names of these functions are not case-sensitive.

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

See also Date and Time Values.

2.11.1 Value Choice

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

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

```
TESTNAMESPACE>set myvar=1

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

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

```
TESTNAMESPACE>set myvar=1

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

2.11.2 Existence Functions

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

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

  For a variable that contains multiple nodes, this function can indicate whether a given node exists, and whether a given node has a value and child nodes.

- To get the value of a variable (if it exists) or get a default value (if not), use the **$GET** function.

2.11.3 List Functions

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

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

  The only supported way to work with a **$LIST** list is to use the ObjectScript list functions. The internal structure of this kind of list is not documented and is subject to change without notice.

- **$LIST** returns a list element or can be used to replace a list element.
- **$LISTLENGTH** returns the number of elements in a list.
- **$LISTFIND** returns the position of a given element, in a given list.

There are additional list functions as well.

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

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

InterSystems IRIS provides other list classes that are *not* equivalent to a list returned by **$LISTBUILD**. These are useful if you prefer to work with classes. For an introduction, see Collection Classes.

2.11.4 String Functions

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

- **$EXTRACT** returns or replaces a substring, using a character count.
- **$FIND** finds a substring by value and returns an integer specifying its end position in the string.
- **$JUSTIFY** returns a right-justified string, padded on the left with spaces.
- **$ZCONVERT** converts a string from one form to another. It supports both case translations (to uppercase, to lowercase, or to title case) and encoding translation (between various character encoding styles).
- **$TRANSLATE** modifies the given string by performing a character-by-character replacement.
- **$REPLACE** performs string-by-string replacement within a string and returns a new string.
- **$PIECE** returns a substring from a character-delimited string (often called a *pieced string*). The following demonstrates how to extract a substring:

  ObjectScript

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

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

  For example:

  ObjectScript

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

2.11.5 Working with Multidimensional Arrays

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

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

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

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

2.11.6 Character Values

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

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

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

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

2.12 Date and Time Values

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

2.12.1 Local Time

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

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

The InterSystems IRIS class library includes data type classes to represent dates in more common formats such as ODBC, and many applications use these instead of $H format. Note that InterSystems supports POSIX time via the %Library.PosixTime data type class, which new applications should use to represent date/time values.

2.12.2 UTC Time

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

2.12.3 Date and Time Conversions

ObjectScript includes functions for converting date and time values.

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

  For example:

  ```
  TESTNAMESPACE>WRITE $ZDATE($HOROLOG,3)
  2010-12-03
  ```

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

  For example:

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

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

2.12.4 Details of the $H Format

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

- The first number is the number of days since December 31st, 1840. That is, day number 1 is January 1st, 1841. This number is always an integer.
- The second number is the number of seconds since midnight on the given day.

  Some functions, such as **$NOW()**, provide a fractional part.

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

2.13 Using Macros and Include Files

As noted earlier, you can define macros and use them later in the same class or routine. More commonly, you define them in include files.

2.13.1 Macros

ObjectScript supports *macros*, which define substitutions. The definition can either be a value, an entire line of code, or (with the **##continue** directive) multiple lines.

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

ObjectScript

```
#define macroname &lt;definition>
```

To refer to a macro, use the following syntax:

```
$$$macroname
```

Or:

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

You use macros to ensure consistency. For example:

ObjectScript

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

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

```
 #define CALL(%C,%A) $$$INTCALL(%C,%A,Quit sc)
```

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

Some of the system classes use macros extensively.

The preprocessor directives are documented in ObjectScript Macros and the Macro Preprocessor in Using ObjectScript.

Note: The Management Portal lists the include files with the routines. Include files are not, however, actually routines because they are not executable.

2.13.2 Include Files

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

After creating an include file, you can do the following:

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

The following shows parts of a system include file:

ObjectScript

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

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

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

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

ObjectScript

```
#include myincludefile
```

To include an include file at the start of a class definition, the directive does not include the pound sign. For example:

```
Include myincludefile
```

Or:

```
Include (myincludefile, yourincludefile)
```

2.14 Using Routines

You can think of a routine as an ObjectScript program. Routines can be written from scratch, or they can be generated automatically when a class is compiled.

2.14.1 Procedures, Functions, and Subroutines

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

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

  A procedure is also called a *procedure block*.
- *Function* (returns a value).
- *Subroutine* (does not return a value).

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

procedure

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

Or:

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

label is the identifier for the procedure.

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

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

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

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

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

function

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

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

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

subroutine

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

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

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

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

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

Variable Availability and Scope has further details.

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

2.14.2 Executing Routines

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

ObjectScript

```
do ^routinename
```

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

ObjectScript

```
do label^routinename
```

Or:

ObjectScript

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

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

ObjectScript

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

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

ObjectScript

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

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

2.14.3 The NEW Command

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

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

ObjectScript

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

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

```
TESTNAMESPACE>do ^demonew

TESTNAMESPACE>write var1
abc
TESTNAMESPACE>write var2

write var2
^
<UNDEFINED> *var2
```

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

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

TESTNAMESPACE>do ^demonew

TESTNAMESPACE>write var2
hello world
```

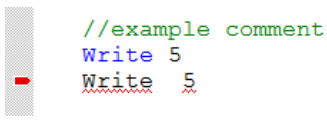
2.15 Potential Pitfalls

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

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

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

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



Similarly, the Terminal displays a syntax error as follows:

```
TESTNAMESPACE>write  5

WRITE  5
       ^
<SYNTAX>
TESTNAMESPACE>
```

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

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

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

  ObjectScript

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

  Similarly, when ObjectScript values are projected to SQL or XML, the values `""` and `$CHAR(0)` are treated differently. For information on the SQL projections of these values, see Null and the Empty String. For information on the XML projections of these values, see Handling Empty Strings and Null Values.

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

  The case-sensitive items include names of most of the elements that you define: routines, variables, classes, properties, and methods. For more details, see Syntax Rules.

- Most command names can be represented by an abbreviated form. Therefore, `WRITE`, `Write`, `write`, `W`, and `w` are all valid forms of the **WRITE** command. For a list, see Table of Abbreviations in the *ObjectScript Reference*.

- For many of the commands, you can include a *postconditional expression* (often simply called a *postconditional*).

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

  For example:

  ObjectScript

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

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

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

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

  ObjectScript

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

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

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

    ObjectScript

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

    InterSystems recommends that you use the block form in all new code.
  - An older line-based form, which does not use curly braces. For example:

    ObjectScript

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

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

  ObjectScript

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

  The class compiler automatically generates compact code of the form shown above (although not necessarily with abbreviated commands as in this example). Sometimes it is useful to look at this generated code, to track down the source of a problem or to understand how something works.
- There are no truly reserved words in ObjectScript, so it is theoretically possible to have a variable named `set`, for example. However, it is prudent to avoid names of commands, functions, SQL reserved words, and certain system items; see Syntax Rules.
- InterSystems IRIS allocates a fixed amount of memory to hold the results of string operations. If a string expression exceeds the amount of space allocated, a <MAXSTRING> error results. See string length limit.

  For class definitions, the string operation limit affects the size of string properties. InterSystems IRIS provides a system object (called a *stream*) that you can use when you need to work with strings that exceed this limit; in such cases, you use the stream interface classes.

2.16 See Also

See the ObjectScript Tutorial.

Also see Classes, Objects, and Persistent Objects and SQL.

3

Classes

This page discusses the basic rules for defining and working with classes in InterSystems IRIS® data platform.

Objects discusses objects and object classes.

Class definitions are not formally part of ObjectScript. Rather, you can use ObjectScript within specific parts of class definitions (particularly within method definitions, where you can also use other implementation languages). The syntax used to define classes is called Class Definition Language, and this syntax is documented separately from ObjectScript.

3.1 Class Names and Packages

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

The package name is simply a string, but if it contains periods, the InterSystems IRIS development tools treat each period-delimited piece as a *subpackage*. Your Integrated Development Environment (IDE) and other tools display these subpackages as a hierarchy of folders, for convenience.

3.2 Basic Contents of a Class Definition

An InterSystems IRIS class definition can include the following items, all known as *class members*:

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

InterSystems IRIS class definitions use Class Definition Language (CDL) to specify the class and its members. You can use either Python or ObjectScript to write the executable code inside of methods.

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

The following shows a simple InterSystems IRIS class definition with methods written in ObjectScript and in Python:

Class/ObjectScript

```
Class MyApp.Main.SampleClass Extends %RegisteredObject
{

Parameter CONSTANTMESSAGE [Internal] = "Hello world!" ;

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

Property MessageCount As %Numeric [Required];

ClassMethod HelloWorld() As %String [ Language = objectscript ]
 {
    Set x=..#CONSTANTMESSAGE
    Return x
 }

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

}
```

Class/Mixed

```
Class MyApp.Main.SampleClass Extends %RegisteredObject
{

Parameter CONSTANTMESSAGE [Internal] = "Hello world!" ;

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

Property MessageCount As %Numeric [Required];

ClassMethod MessageWrapper() As %String [ Language = objectscript ]
{
     return ..#CONSTANTMESSAGE
}

ClassMethod HelloWorld() As %String [ Language = python ]
 {
    import iris
    x = iris.cls("MyApp.Main.SampleClass").MessageWrapper()
    return x
 }

Method WriteIt() [ ServerOnly = 1, Language = python ]
{
    import iris
    CONSTANTMESSAGE = self.MessageWrapper()
    count = self.MessageCount
    print()
    for i in range(count):
        print(CONSTANTMESSAGE, self.VariableMessage)
}

}
```

Note the following points:

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

  Your IDE and other user interfaces treat each package as a folder.

- Extends is a compiler keyword.

  The Extends keyword specifies that this class is a subclass of %RegisteredObject, which is a system class provided for object support. This example class extends only one class, but it is possible to extend multiple other classes. Those classes, in turn, can extend other classes.

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

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

  You must access class parameters via ObjectScript. In the Python version of this class, we use the ObjectScript class method **MessageWrapper()** to return the value of the parameter.

- You can access any class method from Python. You can use the `iris.cls("Package.Class").classMethodName()` syntax in all contexts, and the `self.classMethodName()` syntax from within a Python instance method. The example shows both syntax forms.

- `VariableMessage` and `MessageCount` are properties. The item after As indicates the types for these properties. InitialExpression and Required are compiler keywords.

  You can access an InterSystems IRIS class property directly from ObjectScript or Python, as in the example.

- `HelloWorld()` is a class method and it returns a string; this is indicated by the item after As.

  This method uses the value of the class parameter.

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

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

  The ServerOnly compiler keyword means that this method will not be projected to external clients.

The following Terminal session shows how we can use this class. Both terminal shells are valid for the ObjectScript and Python versions of the class.

ObjectScript Shell

```
TESTNAMESPACE>write ##class(MyApp.Main.SampleClass).HelloWorld()
Hello world!
TESTNAMESPACE>set x=##class(MyApp.Main.SampleClass).%New()

TESTNAMESPACE>set x.MessageCount=3

TESTNAMESPACE>do x.WriteIt()

Hello world! How are you?
Hello world! How are you?
Hello world! How are you?
```

Python Shell

```
>>> print(iris.cls("MyApp.Main.SampleClass").HelloWorld())
Hello world!
>>> x=iris.cls("MyApp.Main.SampleClass")._New()
>>> x.MessageCount=3
>>> x.WriteIt()

Hello world! How are you?
Hello world! How are you?
Hello world! How are you?
```

3.3 Shortcuts for Calling Class Methods

When calling class methods using ObjectScript, you can omit the package (or the higher level packages) in the following scenarios:

- The reference is within a class, and the referenced class is in the same package or subpackage.
- The reference is within a class, and the class uses the IMPORT directive to import the package or subpackage that contains the referenced class.
- The reference is within a method, and the method uses the IMPORT directive to import the package or subpackage that contains the referenced class.

When calling class methods from ObjectScript or Python, you can omit the package (or higher level packages) in the following scenarios:

- You are referring to a class in the %Library package, which is specially handled. You can refer to the class %Library.ClassName as %ClassName. For example, you can refer to %Library.String as %String.
- You are referring to a class in the User package, which is specially handled. For example, you can refer to User.MyClass as MyClass.

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

In all other cases, you must always use the full package and class name to call a class method.

3.4 Class Parameters

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

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

You can define parameters in an InterSystems IRIS class that contains ObjectScript methods, Python methods, or a combination of the two. The following shows a class with several parameters:

Class Definition

```
Class GSOP.DivideWS Extends %SOAP.WebService
{

Parameter USECLASSNAMESPACES = 1;

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

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

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

 ///further details omitted
}
```

Note: Class parameters can also be expressions, which can be evaluated either at compile time or runtime. For more information, see Defining and Referring to Class Parameters.

3.5 Properties

Formally, there are two kinds of properties in InterSystems IRIS:

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

  The word *property* often refers just to properties that are attributes, rather than properties that hold associations.
- Relationships, which hold associations between objects.

You can define properties in a class containing ObjectScript methods, Python methods, or a combination of the two. However, you cannot access relationships from Python methods. This section shows a sample class that contains property definitions that show some of these variations:

Class Definition

```
Class MyApp.Main.Patient Extends %Persistent
{

Property PatientID As %String [Required];

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

Property BirthDate As %Date;

Property Age As %Numeric [Transient];

Property MyTempArray [MultiDimensional];

Property PrimaryCarePhysician As Doctor;

Property Allergies As list Of PatientAllergy;

Relationship Diagnoses As PatientDiagnosis [ Cardinality = children, Inverse = Patient ];
}
```

Note the following:

- In each definition, the item after `As` is the type of the property. Each type is a class. The syntax `As List Of` is shorthand for a specific collection class.

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

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

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

3.5.1 Specifying Property Keywords

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

Required

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

InitialExpression

Specifies an initial value for the property. By default, properties have no initial value. Subclasses inherit the value of the InitialExpression keyword and can override it. The value specified must be a valid ObjectScript expression.

Transient

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

Private

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

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

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

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

Calculated

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

MultiDimensional

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

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

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

3.6 Properties Based on Data Types

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

3.6.1 Data Type Classes

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

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

Each data type class has the following features:

- It specifies values for compiler keywords. For a property, a compiler keyword can do things like the following:
  - Make the property required
  - Specify an initial value for the property
  - Control how the property is projected to SQL, ODBC, and Java clients
- It specifies values for parameters that affect the details such as the following:
  - Maximum and minimum allowed logical value for the data type
  - Maximum and minimum number of characters the string can contain
  - Number of digits following the decimal point
  - Whether to truncate the string if it exceeds the maximum number of characters
  - Display format
  - How to escape any special XML or HTML characters
  - Enumerated lists of logical values and display values to use in any user interface
  - Pattern that the string must match (automatically uses the InterSystems IRIS pattern-matching operator)

  - Whether to respect or ignore the UTC time zone when importing or exporting to XML

- It provides a set of methods to translate literal data among the stored (on disk), logical (in memory), and display formats.

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

Class Definition

```
Class MyApp.MyType Extends %Library.String
{

/// The maximum number of characters the string can contain.
Parameter MAXLEN As INTEGER = 2000;

}
```

3.6.2 Overriding Parameters of Data Type Classes

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

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

Class Member

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

For this property, the maximum allowed value is 10.

(Internally, this works because the validation methods for the data type classes are method generators; the parameter value you provide is used when the compiler generates code for your class.

Similarly, every property of type %String has a collation type, which determines how values are ordered (such as whether capitalization has effects or not). The default collation type is SQLUPPER.

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

Class Member

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

3.6.3 Using Other Property Methods

Properties have a number of methods associated with them automatically. These methods are generated by the data type classes and can be accessed from ObjectScript.

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 name of each generated method is the property name concatenated with the name of the method from the inherited class. You can access these generated methods from ObjectScript, as in the example below. You can access the same

information from Python by calling the associated method directly from the inherited class. For example, some of the methods associated with the %Date class and therefore the DOB property are:

ObjectScript

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

Python

```
x = iris.cls("%Date").IsValid(person.DOB)
print(iris.cls("%Date").LogicalToDisplay(person.DOB))
```

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

3.7 Methods

There are two kinds of methods: instance methods and class methods (called static methods in other languages).

3.7.1 Specifying Method Keywords

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

Language

In InterSystems IRIS, methods can be written in ObjectScript or Python. To specify which language you will write a method in, use the following syntax:

Method/ObjectScript

```
Method MyMethod() [ Language = objectscript ]
{
    // implementation details written in ObjectScript
}
```

Method/Python

```
Method MyMethod() [ Language = python ]
{
    # implementation details written in Python
}
```

If a method does not use the Language keyword the compiler will assume that the method is written in ObjectScript.

You must write the method's language in all lowercase letters, as in the example.

Private

This keyword specifies that the method is private and can only be used with ObjectScript methods. Subclasses inherit the value of the Private keyword and cannot override it.

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

- It can only be accessed by methods of the class to which it belongs.
- It does not appear in the InterSystems Class Reference.

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

Other languages often call such methods *protected methods*.

3.7.2 References to Other Class Members

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

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

  ObjectScript

  ```
  ..#PARAMETERNAME
  ```

  Python

  ```
  # technique 1
  iris.cls("Package.Class")._GetParameter("PARAMETERNAME")

  # technique 2
  objectinstance._GetParameter("PARAMETERNAME")
  ```

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

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

  ObjectScript

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

  Python

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

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

- To refer to another class method, use the following syntax:

  ObjectScript

  ```
  ..classmethodname(arguments)
  ```

  Python

  ```
  # technique 1
  iris.cls("Package.Class").classmethodname(arguments)

  # technique 2
  iris.cls(__name__).classmethodname(arguments)
  ```

  Note that you cannot use the Python *self* syntax to access a class method. Instead, you can use the `__name__` property to obtain the name of the current class, as shown in the above example.

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

  ObjectScript

  ```
  ..PropertyName
  ```

  Python

  ```
  self.PropertyName
  ```

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

ObjectScript

```
..PropertyNameA.PropertyNameB
```

Python

```
self.PropertyNameA.PropertyNameB
```

The syntax used in the ObjectScript examples is known as *dot syntax*.

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

ObjectScript

```
Do ..PropertyName.MyMethod()
```

Python

```
self.PropertyName.MyMethod()
```

3.7.3 References to Methods of Other Classes

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

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

  ObjectScript

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

  Python

  ```
  iris.cls("Package.Class").MethodName(arguments)
  ```

  For example:

  ObjectScript

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

  Python

  ```
  x=iris.cls("Util.Utils").GetToday()
  ```

  You can also invoke a class method without accessing its return value as follows:

  ObjectScript

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

  Python

  ```
  iris.cls("Util.Utils").DumpValues()
  ```

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

- To invoke an instance method, create an instance and then use an expression like the following in either ObjectScript or Python to invoke the method and access its return value:

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

  For example:

  ObjectScript

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

  Python

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

  You can also invoke an instance method without accessing its return value by calling the method as follows:

  ObjectScript

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

  Python

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

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

3.7.4 References to Current Instance

Within an instance method, sometimes it is necessary to refer to the current instance itself, rather than to a property or method of the instance. For example, you might need to pass the current instance as an argument when invoking some other code.

In ObjectScript, use the special variable **$THIS** to refer to the current instance. In Python, use the variable **self** to refer to the current instance.

For example:

ObjectScript

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

Python

```
sc=header.ProcessService(self)
```

3.7.5 Method Arguments

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

For instance, here is the partial definition of a method that takes three arguments. This is valid syntax for both ObjectScript and Python methods within InterSystems IRIS classes:

Class Member

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

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

Note: If a method is defined in Python and has any arguments with default values, then these arguments must be at the end of the argument list to avoid a compilation error.

3.7.5.1 Skipping Arguments

When invoking a method you can skip arguments if there are suitable defaults for them. ObjectScript and Python each have their own syntax to skip arguments.

In ObjectScript, you can skip over an argument by providing no value for that argument and maintaining the comma structure. For example, the following is valid:

ObjectScript

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

In an InterSystems IRIS class, a Python method's signature must list the required arguments first, followed by any arguments with default values.

When calling the method, you must provide arguments in the order of the method's signature. Therefore, once you skip an argument you must also skip all arguments following it. For example, the following is valid:

Member/Python

```
ClassMethod Skip(a1, a2 As %Integer = 2, a3 As %Integer = 3) [ Language = python ]
{
    print(a1, a2, a3)
}

TESTNAMESPACE>do ##class(mypackage.myclass).Skip(1)
1 2 3
```

3.7.5.2 Passing Variables by Value or by Reference

When you invoke a method, you can pass values of variables to that method either by value or by reference.

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

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

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

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

To pass an argument by reference in ObjectScript, place a period before the variable name when invoking the method. In Python, use `iris.ref()` on the value you want to pass and call the method on the reference. Both of these are shown in the following example:

ObjectScript

```
 Do MyMethod(arg1, .arg2, .arg3)
```

Python

```
arg2=iris.ref("peanut butter")
arg3=iris.ref("jelly")
MyMethod(arg1,arg2,arg3)
```

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

3.7.5.3 Variable Numbers of Arguments

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

Method/ObjectScript

```
ClassMethod MultiArg(Arg1... As %List) [ Language = objectscript ]
{
 Set args = $GET(Arg1, 0)
 Write "Invocation has ",
     args,
     " element",
     $SELECT((args=1):"", 1:"s"), !
 For i = 1 : 1 : args
 {
     Write "Argument[", i , "]: ", $GET(Arg1(i), "<NULL>"), !
 }
}
```

Method/Python

```
ClassMethod MultiArg(Arg1... As %List) [ Language = Python ]
{
    print("Invocation has", len(Arg1), "elements")
    for i in range(len(Arg1)):
        print("Argument[" + str(i+1) + "]: " + Arg1[i])
}
```

3.7.5.4 Specifying Default Values

To specify an argument's default value in either an ObjectScript or a Python method, use the syntax as shown in the following example:

Class Member

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

When a method is invoked, it uses its default values (if specified) for any missing arguments. If a method is written in Python, then any arguments with default values must be defined at the end of the argument list.

In ObjectScript, another option is to use the **$GET** function to set a default value. For example:

Class Member

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

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

3.8 Method Generators

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

3.9 Class Queries

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

Class Member

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

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

3.10 XData Blocks

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

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

- WS-Policy support for InterSystems IRIS web service services and web clients. See *Creating Web Services and Web Clients*. In this case, an XData block describes the security policy.
- In Business Intelligence, you use XData blocks to define cubes, subject areas, KPIs, and other elements.

For more details, see Defining and Using XData Blocks.

3.11 Macros and Include Files in Class Definitions

In an InterSystems IRIS class definition, you can define macros in an ObjectScript method and use them in that method. More often, however, you define them in an include file, which you can include at the start of any class definition. For example:

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

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

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

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

3.12 Inheritance Rules in InterSystems IRIS

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

Note that InterSystems IRIS classes cannot inherit from classes defined in Python (meaning a class definition contained in a .py file) and vice versa.

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

3.12.1 Inheritance Order

InterSystems IRIS uses the following rules for inheritance order:

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

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

3.12.2 Primary Superclass

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

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

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

For a persistent class, the primary superclass is especially important; see Classes and Extents.

3.12.3 Most-Specific Type Class

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

3.12.4 Overriding Methods

A class inherits methods (both class and instance methods) from its superclass or superclasses, which you can override. If you do so, you must ensure that the signature in your method definition matches the signature of the method you are overriding. Each argument of the subclass method must use the same data type as the superclass method's argument, or a subclass of that data type. The method in the subclass can, however, specify additional arguments that are not defined in the superclass.

You can override a method written in ObjectScript with a Python method and vice versa as long as the method signatures match.

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

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

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

3.13 See Also

For more information on these topics, see the following resources:

- *Defining and Using Classes* describes how to define classes and class members in InterSystems IRIS.
- *Class Definition Reference* provides reference information for the compiler keywords that you use in class definitions.
- The InterSystems Class Reference provides details on all non-internal classes provided with InterSystems IRIS.

4

Objects

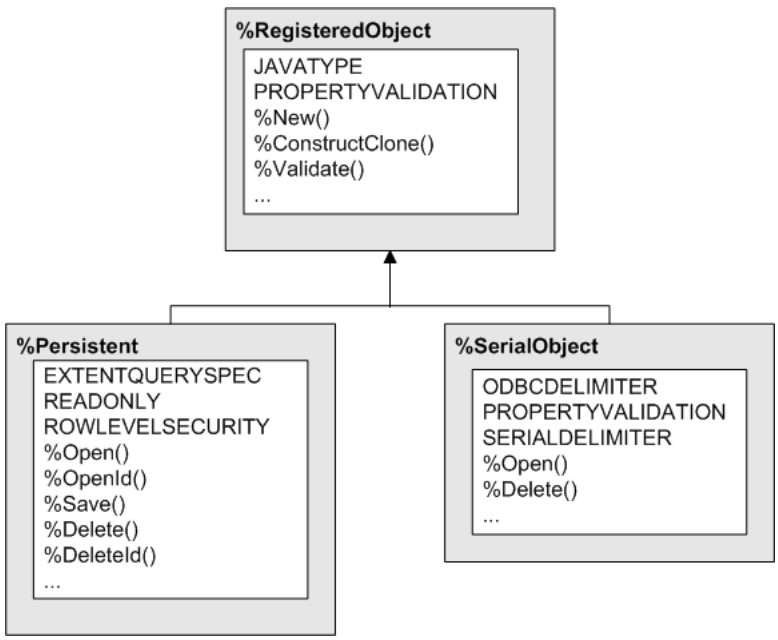
This page discusses objects in InterSystems IRIS® data platform.

The code samples shown in this page use classes from the Samples-Data sample (https://github.com/intersystems/Samples-Data). InterSystems recommends that you create a dedicated namespace called SAMPLES (for example) and load samples into that namespace. For the general process, see *Downloading Samples for Use with InterSystems IRIS*.

4.1 Introduction to InterSystems IRIS Object Classes

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

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



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

- A *registered object* is an instance of %RegisteredObject or a subclass. You can create these objects but you cannot save them. The other two classes inherit from %RegisteredObject and thus include all the parameters, methods, and so on of that class.
- A *persistent object* is an instance of %Persistent or a subclass. You can create, save, open, and delete these objects.

  A persistent class is automatically projected to a table that you can access via InterSystems SQL.
- A *serial object* is an instance of %SerialObject or a subclass. A serial class is meant for use as a property of another object. You can create these objects, but you cannot save them or open them independently of the object that contains them.

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

Note: Via the classes %DynamicObject and %DynamicArray, InterSystems IRIS also provides the ability to work with objects and arrays that have no schema; for details, see *Using JSON*.

4.2 Basic Features of Object Classes

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

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

  For example:

  ObjectScript

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

  Python

  ```
  myobj = iris.cls("Sample.Person")._New()
  ```

  Python method names cannot include a percent sign (%). You can call any ObjectScript method that contains the % character from Python by replacing it with an underscore (_), as in the example.
- You can use properties.

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

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

  Class Member

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

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

ObjectScript

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

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

Python

```
import iris
myaddress=iris.cls("Sample.Address")._New()
myaddress.City="Louisville"
myaddress.Street="15 Winding Way"
myaddress.State="Georgia"

myperson=iris.cls("Sample.Person")._New()
myperson.Home=myaddress
```

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

Method/ObjectScript

```
Method PrintPerson() [ Language = objectscript ]
{
 Write !, "Name: ", ..Name
}
```

Method/Python

```
Method PrintPerson() [ Language = python ]
{
    print("\nName:", self.Name)
}
```

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

ObjectScript

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

Python

```
myobj.PrintPerson()
```

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

  In contrast, when you work at the routine level and do not use classes, your code must include logic to check the type and other input requirements.
- You can define *callback methods* to add additional custom behavior when objects are created, modified, and so on.

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

Method/ObjectScript

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

Method/Python

```
Method %OnNew() As %Status [ Language = python ]
{
    print("hi there")
    return True
}
```

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

4.3 OREFs

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

- In the Terminal, the content of an OREF depends on the language in use:
  - In ObjectScript, you see a string that consists of a number, followed by an at sign (@), followed by the name of the class.
  - In Python, you see a string containing the class name and an 18 character unique location in memory.

  For example:

ObjectScript Shell

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

TESTNAMESPACE>w myobj
3@Sample.Person
```

Python Shell

```
>>> myobj=iris.cls("Sample.Person")._New()
>>> print(myobj)
<iris.Sample.Person object at 0x000001A1E52FFD20>
```

- InterSystems IRIS returns an error if you do not use an OREF where one is expected or you use one with an incorrect type. This error is different from the ObjectScript terminal and the Python terminal:

ObjectScript Shell

```
TESTNAMESPACE>set x=2

TESTNAMESPACE>set x.Name="Fred Parker"

SET x.Name="Fred Parker"
^
<INVALID OREF>
```

Python Shell

```
>>> x=2
>>> x.Name="Fred Parker"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'int' object has no attribute 'Name'
```

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

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

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

ObjectScript Shell

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

Python Shell

```
>>> testthis="<iris.Sample.Person object at 0x000001A1E52FFD20>"
```

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

Note: For persistent classes, methods such as **%OpenId()** also return OREFs.

4.4 Stream Interface Classes

InterSystems IRIS allocates a fixed amount of space to hold the results of string operations. If a string expression exceeds the amount of space allocated, a <MAXSTRING> error results; see string length limit.

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

You can use stream fields with InterSystems SQL, with some restrictions. For details and a more complete introduction, see *Defining and Using Classes*; also see the InterSystems Class Reference for these classes.

Note: You cannot use ObjectScript stream fields with Python.

4.4.1 Stream Classes

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

- The %Stream.FileCharacter and %Stream.FileBinary classes are used for streams written to external files.

  (Binary streams contain the same kind of data as type %Binary, and can hold large binary objects such as pictures. Character streams contain the same kind of data as type %String, and are intended for storing large amounts of text.)

- The %Stream.GlobalCharacter and %Stream.GlobalBinary classes are used for streams stored in globals.

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

4.4.2 Example

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

ObjectScript

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

4.5 Collection Classes

When you need a container for sets of related values, you can use $LIST format lists and multidimensional arrays.

If you prefer to work with classes, InterSystems IRIS provides list classes and array classes; these are called *collections*. For more details on collections, see Working with Collections.

4.5.1 List and Array Classes for Use As Standalone Objects

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

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

Elements in a list are ordered sequentially. Their positions in a list can be accessed using integer key ranging from 1 to N, where N is the position of the last element.

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

ObjectScript

```
set Colors = ##class(%Library.ListOfDataTypes).%New()
do Colors.Insert("Red")
do Colors.Insert("Green")
do Colors.Insert("Blue")

write "Number of list items: ", Colors.Count()
write !, "Second list item: ", Colors.GetAt(2)

do Colors.SetAt("Yellow",2)
write !, "New second item: ", Colors.GetAt(2)

write !, "Third item before insertion: ", Colors.GetAt(3)
do Colors.InsertAt("Purple",3)
write !, "Number of items after insertion: ", Colors.Count()
write !, "Third item after insertion: ", Colors.GetAt(3)
write !, "Fourth item after insertion: ", Colors.GetAt(4)
```

```
do Colors.RemoveAt(3)
write "Number of items after removing item 3: ", Colors.Count()

write "List items:"
for i = 1:1:Colors.Count() write Colors.GetAt(i),!
```

Python

```
import iris

Colors=iris.cls("%Library.ListOfDataTypes")._New()
Colors.Insert("Red")
Colors.Insert("Green")
Colors.Insert("Blue")

print("Number of list items:", Colors.Count())
print("Second list item:", Colors.GetAt(2))

Colors.SetAt("Yellow",2)
print("New second item: ", Colors.GetAt(2))

print("Third item before insertion: ", Colors.GetAt(3))
Colors.InsertAt("Purple",3)
print("Number of items after insertion: ", Colors.Count())
print("Third item after insertion: ", Colors.GetAt(3))
print("Fourth item after insertion: ", Colors.GetAt(4))

Colors.RemoveAt(3)
print("Number of items after removing item 3: ", Colors.Count())

print("List items:")
for i in range(1, Colors.Count() + 1) print(Colors.GetAt(i))
```

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

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

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

ObjectScript

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

Python

```
import iris
ItemArray=iris.cls("%Library.ArrayOfDataTypes")._New()
ItemArray.SetAt("example item", "alpha")
ItemArray.SetAt("another item", "beta")
ItemArray.SetAt("yet another item", "gamma")
ItemArray.SetAt("still another item", "omega")
print("Number of items in this array:", ItemArray.Count())
print("Item that has the key gamma:", ItemArray.GetAt("gamma"))
```

The **SetAt()** method adds items to the array, where the first argument is the element to be added and the second argument is the key. Array elements are ordered by key, with numeric keys first, sorted from smallest to largest, and string keys next, sorted alphabetically with uppercase letters coming before lowercase letters. For example: -2, -1, 0, 1, 2, A, AA, AB, a, aa, ab.

4.5.2 List and Arrays as Properties

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

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

Class Member

```
Property MyProperty as list of Classname;
```

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

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

Class Member

```
Property MyProperty as array of Classname;
```

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

4.6 Useful ObjectScript Functions

ObjectScript provides the following functions for use with object classes:

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

  ```
  TESTNAMESPACE>set class="Sample.Person"

  TESTNAMESPACE>set obj=$CLASSMETHOD(class,"%OpenId",1)

  TESTNAMESPACE>w obj.Name
  Van De Griek,Charlotte M.
  ```

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

  ### ObjectScript

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

  The other functions are useful in similar scenarios.

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

  ```
  TESTNAMESPACE>set obj=##class(Sample.Person).%OpenId(1)

  TESTNAMESPACE>do $METHOD(obj,"PrintPerson")

  Name: Van De Griek,Charlotte M.
  ```

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

  ```
  TESTNAMESPACE>set obj=##class(Sample.Person).%OpenId(2)

  TESTNAMESPACE>write $property(obj,"Name")
  Edison,Patrick J.
  ```

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

```
TESTNAMESPACE>set obj=##class(Sample.Person).%OpenId(2)

TESTNAMESPACE>write $parameter(obj,"EXTENTQUERYSPEC")
Name,SSN,Home.City,Home.State
```

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

```
TESTNAMESPACE>set obj=##class(Sample.Person).%OpenId(1)

TESTNAMESPACE>write $CLASSNAME(obj)
Sample.Person
```

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

4.7 See Also

For more information on the topics covered in this page, see the following resources:

- *Defining and Using Classes* describes how to define classes and class members in InterSystems IRIS.
- *Class Definition Reference* provides reference information for the compiler keywords that you use in class definitions.
- The InterSystems Class Reference provides details on all non-internal classes provided with InterSystems IRIS.

5

Persistent Objects and InterSystems IRIS SQL

A key feature in InterSystems IRIS® data platform is its combination of object technology and SQL. You can use the most convenient access mode for any given scenario. This page describes how InterSystems IRIS provides this feature and gives an overview of your options for working with stored data.

The ObjectScript samples shown in this page are from the Samples-Data sample (https://github.com/intersystems/Samples-Data). InterSystems recommends that you create a dedicated namespace called SAMPLES (for example) and load samples into that namespace. For the general process, see *Downloading Samples for Use with InterSystems IRIS*.

5.1 Introduction

InterSystems IRIS is a multi-model data platform combined with an object-oriented programming language. As a result, you can write flexible code that does all of the following:

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

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

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

5.2 InterSystems SQL

InterSystems IRIS provides an implementation of SQL, known as InterSystems SQL. You can use InterSystems SQL within methods and within routines.

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

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

5.2.1 Using SQL from ObjectScript

You can execute SQL from ObjectScript using either or both of the following ways:

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

  ObjectScript

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

  You can use dynamic SQL in ObjectScript methods and routines.

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

  ObjectScript

  ```
  &sql(SELECT COUNT(*) INTO :myvar FROM Sample.Person)
     IF SQLCODE<0 {WRITE "SQLCODE error ",SQLCODE," ",%msg  QUIT}
     ELSEIF SQLCODE=100 {WRITE "Query returns no results"  QUIT}
  WRITE myvar
  ```

  You can use embedded SQL in ObjectScript methods and routines.

5.2.2 Using SQL from Python

You can execute SQL from Python using either or both of the following ways:

- You can execute the SQL query directly, as in the following example:

  Python

  ```
  import iris
  rset = iris.sql.exec("SELECT TOP 5 Name, DOB FROM Sample.Person")
  for row in rset:
      print(row)
  ```

- You can also prepare the SQL query first, then execute it, as in the following example:

  Python

  ```
  import iris
  statement = iris.sql.prepare("SELECT TOP 5 Name, DOB FROM Sample.Person")
  rset = statement.execute()
  for row in rset:
      print(row)
  ```

You can use either of these approaches to execute SQL queries in the Python terminal or in Python methods.

5.2.3 Object Extensions to SQL

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

One of the most interesting of these extensions is ability to follow object references using the implicit join operator (`->`), sometimes referred to as "arrow syntax." For example, suppose you have a Vendor class that refers to two other classes: Contact and Region. You can refer to properties of the related classes using the implicit join operator:

SQL

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

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

5.3 Special Options for Persistent Classes

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

- Ability to use methods to open, save, and delete objects.

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

  When you open an object instance and you refer to an object-valued property, the system automatically opens that object as well. This process is referred to as *swizzling*. Then you can work with that object as well. In the example below, when the Sample.Person object is opened, the corresponding Sample.Address object is swizzled:

  #### ObjectScript

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

  #### Python

  ```
  import iris
  person=iris.cls("Sample.Person")._OpenId(10)
  person.Name="Andrew Park"
  person.Address.City="Birmingham"
  person._Save()
  ```

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

- Ability to use a default query (the Extent query) that is an SQL result set that contains the data for the objects of this class. By default, the Extent query returns the existing IDs in the extent. It can be modified to return more columns.

  In this class (or in other classes), you can define additional queries.

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

  A relationship is a special type of object-valued property that defines how two or more object instances are associated with each other. Every relationship is two-sided: for every relationship definition, there is a corresponding inverse

relationship that defines the other side. InterSystems IRIS automatically enforces referential integrity of the data, and any operation on one side is immediately visible on the other side. Relationships automatically manage their in-memory and on-disk behavior. They also provide superior scaling and concurrency over object collections (see Collection Classes).

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

  Indexes 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.

  Indexes 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, indexes can store additional data that is frequently requested by queries based on the sorted properties. By including additional data as part of an index, you can greatly enhance the performance of the query that uses the index; when the query uses the index to generate its result set, it can do so without accessing the main data storage facility.
- Ability to define triggers in these classes to control what occurs when rows are inserted, modified, or deleted.
- Ability to project methods and class queries as SQL stored procedures.
- Ability to fine-tune the projection to SQL (for example, specifying the table and column names as seen in SQL queries).
- Ability to fine-tune the structure of the globals that store the data for the objects.

Note: You cannot define relationships, foreign keys, or indexes in Python.

5.4 SQL Projection of Persistent Classes

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

5.4.1 Demonstration of the Object-SQL Projection

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

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

Note the following points:

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

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

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

ObjectScript Shell

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

SAMPLES>write person.Name
Newton,Dave R.
SAMPLES>write person.FavoriteColors.Count()
1
SAMPLES>write person.FavoriteColors.GetAt(1)
Red
SAMPLES>write person.SSN
384-10-6538
```

Python Shell

```
>>> person=iris.cls("Sample.Person")._OpenId(1)
>>> print(person.Name)
Newton,Dave R.
>>> print(person.FavoriteColors.Count())
1
>>> print(person.FavoriteColors.GetAt(1))
Red
>>> print(person.SSN)
384-10-6538
```

These are the same values that we see via SQL.

5.4.2 Basics of the Object-SQL Projection

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

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

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

5.4.3 Classes and Extents

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

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

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

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

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

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

5.5 Object IDs

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

- **%DeleteId()**
- **%ExistsId()**
- **%OpenId()**

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

5.5.1 How an ID Is Determined

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

Any ID is unique within its extent.

The ID for an object is determined as follows:

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

  ```
  parentID||childID
  ```

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

```
104||3
```

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

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

  ```
  SKU-447
  ```

  Also, the property value cannot be changed.

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

  ```
  CATEGORY12||SUBCATEGORYA
  ```

  Also, these property values cannot be changed.

5.5.2 Accessing an ID

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

ObjectScript Shell

```
SAMPLES>set person=##class(Sample.Person).%OpenId(2)

SAMPLES>write person.%Id()
2
```

Python Shell

```
>>> person = iris.cls("Sample.Person")._OpenId(2)
>>> print(person._Id())
2
```

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

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

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

5.6 Storage

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

5.6.1 A Look at a Storage Definition

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

```
<Storage name="Default">
<Data name="PersonDefaultData"><Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Name</Value>
</Value>
<Value name="3">
<Value>SSN</Value>
</Value>
<Value name="4">
<Value>DOB</Value>
</Value>
...
</Storage>
```

5.6.2 Globals Used by a Persistent Class

The storage definition includes several elements that specify the globals in which the data is stored:

```
<DataLocation>^Sample.PersonD</DataLocation>
<IdLocation>^Sample.PersonD</IdLocation>
<IndexLocation>^Sample.PersonI</IndexLocation>
...
<StreamLocation>^Sample.PersonS</StreamLocation>
```

By default, with default storage:

- The class data is stored in the *data global* for the class. Its name starts with the complete class name (including package name). A `D` is appended to the name. For example: `Sample.PersonD`
- The index data is stored in the *index global* for the class. Its name starts with the class name and ends with an `I`. For example: `Sample.PersonI`
- Any saved stream properties are stored in the *stream global* for the class. Its name starts with the class name and ends with an `S`. For example: `Sample.PersonS`

Important: If the complete class name is long, the system automatically uses a hashed form of the class name instead. So when you view a storage definition, you might sometimes see global names like `^package1.pC347.VeryLongCla4F4AD`. If you plan to work directly with the data global for a class for any reason, make sure to examine the storage definition so that you know the actual name of the global.

For more information on how global names are determined, see Globals in *Defining and using Classes*.

5.6.3 Notes

Note the following points:

- Never redefine or delete storage for a class that has stored data. If you do so, you will have to recreate the storage manually, because the new default storage created when you next compile the class might not match the required storage for the class.
- During development, you may want to reset the storage definition for a class. You can do this if you also delete the data and later reload or regenerate it.
- By default, as you add and remove properties during development, the system automatically updates the storage definition, via a process known as *schema evolution*.

  The exception is if you use a non-default storage class for the <Type> element. The default is %Storage.Persistent; if you do not use this storage class, InterSystems IRIS does not update the storage definition.

5.7 Options for Creating Persistent Classes and Tables

To create a persistent class and its corresponding SQL table, you can do any of the following:

- Use your IDE to define a class based on %Persistent. When you compile the class, the system creates the table.
- In the Management Portal, you can use the Data Migration Wizard, which reads an external table, prompts you for some details, generates a class based on %Persistent, and then loads records into the corresponding SQL table.

  You can run the wizard again later to load more records, without redefining the class.
- In the Management Portal, you can use the Link Table Wizard, which reads an external table, prompts you for some details, and generates a class that is linked to the external table. The class retrieves data at runtime from the external table.
- In InterSystems SQL, use CREATE TABLE or other DDL statements. This also creates a class.
- In the Terminal (or in code), use the **CSVTOCLASS()** method of %SQL.Util.Procedures. For details, see the Class Reference for %SQL.Util.Procedures.

5.8 Accessing Data

To access, modify, and delete data associated with a persistent class, your code can do any or all of the following:

- Open instances of persistent classes, modify them, and save them.
- Delete instances of persistent classes.
- Use embedded SQL.
- Use dynamic SQL (the SQL statement and result set interfaces).
- Use SQL from Python.
- Use low-level commands and functions for direct global access. Note that this technique is not recommended *except* for retrieving stored values, because it bypasses the logic defined by the object and SQL interfaces.

InterSystems SQL is suitable in situations like the following:

- You do not initially know the IDs of the instances to open but will instead select an instance or instances based on input criteria.
- You want to perform a bulk load or make bulk changes.

- You want to view data but not open object instances.

  (Note, however, that when you use object access, you can control the degree of concurrency locking. If you know that you do not intend to change the data, you can use minimal concurrency locking.)

- You are fluent in SQL.

Object access is suitable in situations like the following:

- You are creating a new object.
- You know the ID of the instance to open.
- You find it more intuitive to set values of properties than to use SQL.

5.9 A Look at Stored Data

This section demonstrates that for any persistent object, the same values are visible via object access, SQL access, and direct global access.

In our IDE, if we view the Sample.Person class, we see the following property definitions:

```
/// Person's name.
Property Name As %String(POPSPEC = "Name()") [ Required ];

...

/// Person's age.<br>
/// This is a calculated field whose value is derived from <property>DOB</property>.
Property Age As %Integer [ details removed for this example ];

/// Person's Date of Birth.
Property DOB As %Date(POPSPEC = "Date()");
```

In the Terminal, we can open a stored object and write its property values:

ObjectScript Shell

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

SAMPLES>w person.Name
Newton,Dave R.
SAMPLES>w person.Age
21
SAMPLES>w person.DOB
58153
```

Python Shell

```
>>> person=iris.cls("Sample.Person")._OpenId(1)
>>> print(person.Name)
Newton, Dave R.
>>> print(person.Age)
21
>>> print(person.DOB)
58153
```

Note that here we see the literal, stored value of the DOB property. We could instead call a method to return the display value of this property:

ObjectScript Shell

```
SAMPLES>write person.DOBLogicalToDisplay(person.DOB)
03/20/2000
```

Python Shell

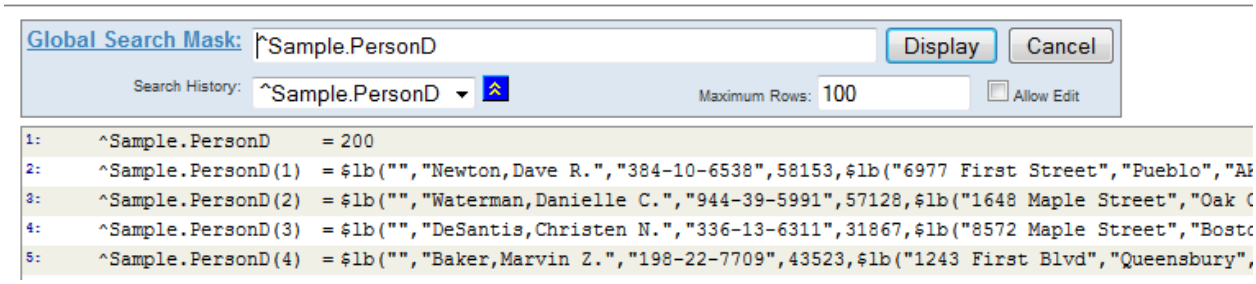
```
>>> print(iris.cls("%Date").LogicalToDisplay(person.DOB))
03/20/2000
```

In the Management Portal, we can browse the stored data for this class, which looks as follows:

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

Notice that in this case, we see the display value for the DOB property. (In the Portal, there is another option to execute queries, and with that option you can control whether to use logical or display mode for the results.)

In the Portal, we can also browse the global that contains the data for this class:



Or, in the Terminal, we can write the value of the global node that contains this instance using ObjectScript:

```
zwrite ^Sample.PersonD("1")
^Sample.PersonD(1)=$lb("","Newton,Dave R.","384-10-6538",58153,$lb("6977 First
Street","Pueblo","AK",63163),
$lb("9984 Second Blvd","Washington","MN",42829),"",$lb("Red"))
```

For reasons of space, the last example contains an added line break.

5.10 Storage of Generated Code for InterSystems SQL

For InterSystems SQL, the system generates reusable code to access the data.

When you first execute an SQL statement, InterSystems IRIS optimizes the query and generates and stores code that retrieves the data. It stores the code in the *query cache*, along with the optimized query text. Note that this cache is a cache of code, not of data.

Later when you execute an SQL statement, InterSystems IRIS optimizes it and then compares the text of that query to the items in the query cache. If InterSystems IRIS finds a stored query that matches the given one (apart from minor differences such as whitespace), it uses the code stored for that query.

You can view the query cache and delete any items in it.

5.11 See Also

For more information on the topics covered in this page, see the following resources:

- *Defining and Using Classes* describes how to define classes and class members in InterSystems IRIS.
- *Class Definition Reference* provides reference information for the compiler keywords that you use in class definitions.
- *Using InterSystems SQL* describes how to use InterSystems SQL and where you can use it.
- *InterSystems SQL Reference* provides reference information on InterSystems SQL.
- *Using Globals* provides details on how InterSystems IRIS stores persistent objects in globals.
- The InterSystems Class Reference has information on all non-internal classes provided by InterSystems IRIS.

6

Namespaces and Databases

This page describes how InterSystems IRIS® data platform organizes data and code.

6.1 Introduction to Namespaces and Databases

In InterSystems IRIS, any code runs in a *namespace*, which is a logical entity. A namespace provides access to data and to code, which is stored (typically) in multiple databases. A *database* is a file — an IRIS.DAT file. InterSystems IRIS provides a set of namespaces and databases for your use, and you can define additional ones.

In a namespace, the following options are available:

- A namespace has a default database in which it stores code; this is the *routines database* for this namespace.

  When you write code in a namespace, the code is stored in its routines database unless other considerations apply. Similarly, when you invoke code, InterSystems IRIS looks for it in this database unless other considerations apply.

- A namespace also has a default database to contain data for persistent classes and any globals you create; this is the *globals database* for this namespace.

  So, for example, when you access data (in any manner), InterSystems IRIS retrieves it from this database unless other considerations apply.

  The globals database can be the same as the routines database, but it is often desirable to separate them for maintainability.

- A namespace has a default database for temporary storage.

- A namespace can include *mappings* that provide access to additional data and code that is stored in other databases. Specifically, you can define mappings that refer to routines, class packages, entire globals, and specific global nodes in non-default databases. (These kinds of mappings are called, respectively, *routine mappings*, *package mappings*, *global mappings*, and *subscript-level mappings*.)

  When you provide access to a database via a mapping, you provide access to only a part of that database. The namespace cannot access the non-mapped parts of that database, not even in a read-only manner.

  Also, it is important to understand that when you define a mapping, that affects only the configuration of the namespace. It does not change the current location of any code or data. Thus when you define a mapping, it is also necessary to move the code or data (if any exists) from its current location to the one expected by the namespace.

  Defining mappings is a database administration task and requires no change to class/table definitions or application logic.

- Any namespace you create has access to most of the InterSystems IRIS code library. This code is available because InterSystems IRIS automatically establishes specific mappings for any namespace you create.

  To find tools for a particular purpose, see the *InterSystems Programming Tools Index*.

- When you define a namespace, you can cause it to be *interoperability-enabled*. This means that you can define a production in this namespace. A production is a program that uses InterSystems IRIS Interoperability features and integrates multiple separate software systems; to read about this, see Introducing Interoperability Productions.

Mappings provide a convenient and powerful way to share data and code. Any given database can be used by multiple namespaces. For example, there are several system databases that all customer namespaces can access, as discussed later in this page.

You can change the configuration of a namespace after defining it, and InterSystems IRIS provides tools for moving code and data from one database to another. Thus you can reorganize your code and data during development, if you discover the need to do so. This makes it possible to reconfigure InterSystems IRIS applications (such as for scaling) with little effort.

6.1.1 Locks, Globals, and Namespaces

Because a global can be accessed from multiple namespaces, InterSystems IRIS provides automatic cross-namespace support for its locking mechanism. A lock on a given global applies automatically to *all* namespaces that use the database that stores the global.

6.2 Database Basics

An InterSystems IRIS *database* is an IRIS.DAT file. You create a database via the Management Portal. Or if you have an existing InterSystems IRIS database, you can configure InterSystems IRIS to become aware of it.

6.2.1 Database Configuration

For any database, InterSystems IRIS requires the following configuration details:

- Logical name for the database.
- Directory in which the IRIS.DAT file resides. When you create a database in the Management Portal, you are prompted to choose or create a subdirectory within the *system manager's directory* (*install-dir*/Mgr), but you can store the database file in any convenient directory.

  Tip: It is convenient to use the same string for the logical name and for the directory that contains the IRIS.DAT file. The system-provided InterSystems IRIS databases follow this convention.

Additional options include the following:

- Default directory to use for file streams used by this database.

  This is important because your users will need write access to this directory; if not, your code will not be able to create file streams.

- Collation of new globals.
- Initial size and other physical characteristics.
- Option to enable or disable *journaling*. Journaling tracks changes made to an InterSystems IRIS database, for up-to-the-minute recovery after a crash or restoring your data during system recovery.

In most cases, it is best to enable journaling. However, you might want to disable journaling for designated temporary work spaces; for example, the IRISTEMP database is not journaled.

- Option to mount this database for read-only use.

  If a user tries to set a global in a read-only database, InterSystems IRIS returns a <PROTECT> error.

In most cases, you can create, delete, or modify database attributes while the system is running.

6.2.2 Database Features

With each database, InterSystems IRIS provides physical integrity guarantees for both the actual data and the metadata that organizes it. This integrity is guaranteed even if an error occurs during writes to the database.

The databases are automatically extended as needed, without manual intervention. If you expect a particular database to grow and you can determine how large it will become, you can "pre-expand" it by setting its initial size to be near the expected eventual size. If you do so, the performance is better.

InterSystems IRIS provides a number of strategies that allow high availability and recoverability. These include:

- Journaling — Introduced earlier.
- Mirroring — Provides rapid, reliable, robust, automatic failover between two InterSystems IRIS systems, making mirroring the ideal automatic failover high-availability solution for the enterprise.
- Clustering — There is full support of clustering on operating systems that provide it.

InterSystems IRIS has a technology for distributing data and application logic and processing among multiple systems. It is called the Enterprise Cache Protocol (ECP). On a multiserver system, a network of InterSystems IRIS database servers can be configured as a common resource, sharing data storage and application processing, with the data distributed seamlessly among them. This provides increased scalability as well as automatic failover and recovery.

6.2.3 Database Portability

InterSystems IRIS databases are portable across platforms and across versions, with the following caveat:

- On different platforms, any file is either *big-endian* (that is, most-significant byte first) or *little-endian* (least-significant byte first).

  InterSystems IRIS provides a utility to convert the byte order of an InterSystems IRIS database; it is called cvendian. This is useful when moving a database among platforms of the two types.

6.3 System-Supplied Databases

InterSystems IRIS provides the following databases:

ENSLIB

Read-only database contains additional code needed for InterSystems IRIS Interoperability features, specifically the ability to create productions, which integrate separate software systems.

If you create a namespace that is interoperability-enabled, that namespace has access to the code in this database.

IRISAUDIT

Read/write database used for audit records. Specifically, when you enable event logging, InterSystems IRIS writes the audit data to this database.

IRISLIB

Read-only database that includes the object, data type, stream, and collection classes and many other class definitions. It also includes the system include files, generated INT code (for most classes), and generated OBJ code.

IRISLOCALDATA

Read/write database that contains items used internally by InterSystems IRIS, such as cached SQL queries and CSP session information.

Note: No customer application should directly interact with the IRISLOCALDATA database. This database is purely for internal use by InterSystems IRIS.

IRISSYS (the *system manager's database*)

Read/write database includes utilities and data related to system management. It is intended to contain specific custom code and data of yours and to preserve that code and data upon upgrades.

This database contains or can contain:

- Users, roles, and other security elements (both predefined items and ones that you add).

  For reasons of security, the Management Portal handles this data differently than other data; for example, you cannot display a table of users and their passwords.
- Data for use by the NLS (National Language Support) classes: number formats, the sort order of characters, and other such details. You can load additional data.
- Your own code and data. To ensure that these items are preserved upon upgrades, use the naming conventions in Custom Items in IRISSYS.

CAUTION: InterSystems does not support moving, replacing, or deleting the IRISSYS database.

The directory that contains this database is the *system manager's directory*. The messages log (messages.log) is written to this directory, as are other log files.

For additional detail on IRISSYS, see Using Resources to Protect Assets.

IRISTEMP

Read/write database used for temporary storage. InterSystems IRIS uses this database, and you can also use it. Specifically, this database contains *temporary globals*. For details, see Temporary Globals and the IRISTEMP Database.

USER

An initially empty read/write database meant for your custom code. You do not have to use this database.

HSCUSTOM, HSLIB, and HSSYS

Databases that provide code for IRIS for Health™ and for HealthShare®. Not available in other products.

HSAALIB, HSCOMMLIB, HSPD, HSPDLIB, HSPILIB, and VIEWERLIB

Databases that provide access to HealthShare® features; applies only to HealthShare. Not available in other products.

HSSYSLOCALTEMP

Read/write database used for temporary storage used internally by InterSystems IRIS. This database is not journaled, not mirrored, and has no public permissions; it is used to store *temporary globals*.

Important: No customer application should directly interact with the HSSYSLOCALTEMP database. This database is purely for internal use by InterSystems IRIS.

6.4 System-Supplied Namespaces

InterSystems IRIS provides the following namespaces:

%SYS

The namespace in which it is possible to execute privileged system code. See %SYS Namespace.

USER

An initially empty namespace meant for your custom code. You do not have to use this namespace.

ENSLIB

A namespace configured to support interoperability productions. That is, this namespace is interoperability-enabled.

HSCUSTOM, HSLIB, and HSSYS

Namespaces that provide access to features for IRIS for Health™ and for HealthShare®. Not available in other products.

HSAALIB, HSCOMMLIB, HSPD, HSPDLIB, HSPILIB, and VIEWERLIB

Namespaces that provide access to HealthShare® features; applies only to HealthShare. Not available in other products.

HSSYSLOCALTEMP

A namespace that provides access to the HSSYSLOCALTEMP database.

6.5 %SYS Namespace

The %SYS namespace provides access to code that should *not* be available in all namespaces — code that manipulates security elements, the server configuration, and so on.

For this namespace, the default routines database and default globals database is IRISSYS. If you follow certain naming conventions, you can create your own code and globals in this namespace and store them in the IRISSYS database; see Custom Items in IRISSYS.

6.6 What Is Accessible in Your Namespaces

When you create a namespace, the system automatically defines mappings for that namespace. As a result, in that namespace, you can use the following items (provided you are logged in as a user with suitable permissions for these items):

- Any class whose package name starts with a percent sign (%). This includes most, but not all, classes provided by InterSystems IRIS.
- All the code stored in the routines database for this namespace.
- All the data stored in the globals database for this namespace.
- Any routine whose name starts with a percent sign.
- Any include file whose name starts with a percent sign.
- Any global whose name starts with a caret and a percent sign (^%). These globals are generally referred to as *percent globals*. Note that via global mappings or subscript level mappings, it is possible to change where percent globals are stored, but that has no effect on their visibility. Percent globals are always visible in all namespaces.
- Your own globals with names that start `^IRIS.TempUser` — for example, `^IRIS.TempUser.MyApp`. If you create such globals, these globals are written to the IRISTEMP database.
- If the namespace is interoperability-enabled, you can use code in the Ens and EnsLib packages. The CSPX and EnsPortal packages are also visible but these are not meant for direct use.

  If a namespace is interoperability-enabled, you can define a production in this namespace. To read about this, see *Introducing Interoperability Productions.*
- Any additional code or data that is made available via mappings defined in this namespace.

Via extended global references, your code can access globals that are defined in other namespaces. For information, see Global Structure.

The InterSystems IRIS security model controls which data and which code any user can access.

6.6.1 System Globals in Your Namespaces

Your namespaces contain additional system globals, which fall into two rough categories:

- System globals that are in all namespaces. These include the globals in which InterSystems IRIS stores your routines, class definitions, include files, INT code, and OBJ code.
- System globals that are created when you use specific InterSystems IRIS features. For example, if you use Analytics in a namespace, the system creates a set of globals for its own internal use.

In most cases, you should not manually write to or delete any of these globals. See Global Naming Conventions.

6.7 Stream Directory

In any given namespace, when you create a file stream, InterSystems IRIS writes a file to a default directory and then later deletes it.

This is important because your users will need write access to this directory; if not, your code will not be able to create file streams.

The default directory is the stream subdirectory of the globals database for this namespace.

6.8 See Also

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

- *Using Globals* has information on working with globals, including using extended references.
- Using cvendian to Convert Between Big-endian and Little-endian Systems has information on cvendian.
- *High Availability Guide* has information on high availability and recoverability.
- *Scalability Guide* has information on distributed caching and sharding.

7

InterSystems IRIS Security

This page provides an overview of InterSystems security, with emphasis on the topics most relevant to programmers who write or maintain InterSystems IRIS® data platform applications.

For more information about security, see About InterSystems Security.

7.1 Security Elements Within InterSystems IRIS

InterSystems security provides a simple, unified security architecture that is based on the following elements:

- *Authentication*. Authentication is how you prove to InterSystems IRIS that you are who you say you are. Without trustworthy authentication, authorization mechanisms are moot — one user can impersonate another and then take advantage of the fraudulently obtained privileges.

  The authentication mechanisms available depend on how you are accessing InterSystems IRIS. InterSystems IRIS has a number of available authentication mechanisms. Some require programming effort.

- *Authorization*. Once a user is authenticated, the next security-related question to answer is what that person is allowed to use, view, or alter. This determination and control of access is known as *authorization*.

  As a programmer, you are responsible for including the appropriate security checks within your code to make sure that a given user has permission to perform a given task.

- *Auditing*. Auditing provides a verifiable and trustworthy trail of actions related to the system, including actions of the authentication and authorization systems. This information provides the basis for reconstructing the sequence of events after any security-related incident. Knowledge of the fact that the system is audited can serve as a deterrent for attackers (because they know they will reveal information about themselves during their attack).

  InterSystems IRIS provides a set of events that can be audited, and you can add others. As a programmer, you are responsible for include the audit logging in your code for your custom events.

- *Database encryption*. InterSystems IRIS database encryption protects data at rest — it secures information stored on disk — by preventing unauthorized users from viewing this information. InterSystems IRIS implements encryption using the AES (Advanced Encryption Standard) algorithm. Encryption and decryption occur when InterSystems IRIS writes to or reads from disk. In InterSystems IRIS, encryption and decryption have been optimized, and their effects are both deterministic and small for any InterSystems IRIS platform; in fact, there is no added time at all for writing to an encrypted database.

  The task of database encryption does not generally require you to write code.

7.2 Secure Communications to and From InterSystems IRIS

When communicating between InterSystems IRIS and external systems, you can use the following additional tools:

- *SSL/TLS configurations*. InterSystems IRIS supports the ability to store a SSL/TLS configuration and specify an associated name. When you need an SSL/TLS connection (for HTTP communications, for example), you programmatically provide the applicable configuration name, and InterSystems IRIS automatically handles the SSL/TLS connection.
- *X.509 certificate storage*. InterSystems IRIS supports the ability to load an X.509 certificate and private key and specify an associated configuration name. When you need an X.509 certificate (to digitally sign a SOAP message, for example), you programmatically provide the applicable configuration name, and InterSystems IRIS automatically extracts and uses the certificate information.

  You can optionally enter the password for the associated private key file, or you can specify this at runtime.
- *Access to a certificate authority (CA)*. If you place a CA certificate of the appropriate format in the prescribed location, InterSystems IRIS uses it to validate digital signatures and so on.

  InterSystems IRIS uses the CA certificate automatically; no programming effort is required.

7.3 InterSystems IRIS Applications

The InterSystems IRIS security model includes *applications*, which are configurations that control authentication, authorization, and other aspects of code use. These apply to user interfaces, external executables, and APIs.

For an overview, see InterSystems IRIS Applications.

For details, see Defining Applications.

7.4 InterSystems Authorization Model

As a programmer, you are responsible for including the appropriate security checks within your code to make sure that a given user has permission to perform a given task. Therefore, it is necessary to become familiar with the InterSystems authorization model, which uses role-based access. Briefly, the terms are as follows:

- *Assets*. Assets are the items being protected. Assets vary widely in nature. The following items are all assets:
  - Each InterSystems IRIS database
  - The ability to connect to InterSystems IRIS using SQL
  - The ability to perform backups
  - Each Analytics KPI class
  - Each application defined in InterSystems IRIS
- *Resources*. A resource is an InterSystems security element that you can associate with one or more assets.

For some assets, the association between an asset and a resource is a configuration option. When you create a database, you specify the associated resource. Similarly, when you create an InterSystems IRIS application, you specify the associated resource.

For other assets, the association is hardcoded. For an Analytics KPI class, you specify the associated resource as a parameter of that class.

For assets and resources that you define, you are free to make the association in either manner — either by hardcoding it or by defining a suitable configuration system.

- *Roles*. A role is an InterSystems security element that specifies a name and an associated set of privileges (possibly quite large). A *privilege* is a *permission* of a specific type (Read, Write, or Use) on a specific resource. For example, the following are privileges:
  - Permission to read a database
  - Permission to write to a table
  - Permission to use an application
- *Usernames*. A username (or a *user*, for short) is an InterSystems security element with which a user logs on to InterSystems IRIS. Each user *belongs to* (or *is a member of*) one or more roles.

Another important concept is *role escalation*. Sometimes it is necessary to temporarily add one or more new roles to a user (programmatically) so that the user can perform a normally disallowed task within a specific context. This is known as *role escalation*. After the user exits that context, you would remove the temporary roles; this is *role de-escalation*.

You define, modify, and delete resources, roles, and users within the Management Portal (provided that you are logged in as a user with sufficient privileges). When you deploy your applications, however, you are more likely to define resources, roles, and starter usernames programmatically, as part of installation; InterSystems IRIS provides ways to do so.

8

InterSystems IRIS Applications

Almost all users interact with InterSystems IRIS® data platform via *applications*, which are configurations that control authentication, authorization, and other aspects of code use. These apply to user interfaces, external executables, and APIs.

This page provides an introduction to help you understand your options as an application developer.

8.1 Types of Applications

Formally there are four types of applications:

- Web applications — these applications connect to InterSystems IRIS via the *Web Gateway*. These are either web-based APIs that provide access to the database (via REST or SOAP) or are user interfaces (HTML pages including your choice of JavaScript libraries).
- Privileged routine applications — these applications are typically executed at the command line and they do not involve the Web Gateway. Examples include routines that are meant for back-end use only for specific users.
- Client applications — these applications invoke external executables and are available only on Windows.
- Document database applications — these applications connect to InterSystems IRIS using the document database.

8.2 Properties of Applications

The properties of applications (that is, the actual configurations) vary by type, so the following list summarizes the most common properties:

- A security resource, whose purpose varies by application type. For example, for web applications and client applications, the resource controls whether users have access to the application; a user must hold the USE permission on the given resource in order to use the application.
- Identifier of the code to execute (the REST dispatch class, the routine to run, the executable to run, and so on).
- Allowed authentication mechanisms.
- Namespace to run the code in (if applicable).
- Options for controlling sessions (for web applications).

- The *application roles* — This is the set of security roles to automatically add to the authenticated user when executing the code. (This configuration option supplements any programmatic role escalation.)
- The *matching roles* — This is a more targeted version of application roles. If the user already has a specific role, you can specify a set of additional roles to automatically add. (This configuration option supplements any programmatic role escalation.)

8.3 How Applications Are Defined

You can define, modify, and applications within the Management Portal (provided that you are logged in as a user with sufficient privileges). When you deploy your applications, however, you are more likely to define applications programmatically as part of installation; InterSystems IRIS provides ways to do so.

8.4 See Also

- Defining Applications, which includes information on programmatic role escalation
- Custom Web APIs and Applications

9

Localization in InterSystems IRIS

This page provides an overview of InterSystems IRIS® data platform support for localization.

9.1 Introduction

InterSystems IRIS supports localization so that you can develop applications for multiple countries or multiple areas.

First, on the client side:

- The Management Portal displays strings in the local language as specified by the browser settings, for a fixed set of languages.
- You can provide localized strings for your own applications as well. See *String Localization and Message Dictionaries*.

On the server side, there are additional considerations:

- InterSystems IRIS provides a set of predefined locales. An InterSystems IRIS *locale* is a set of metadata that specify the user language, currency symbols, formats, and other conventions for a specific country or geographic region.

  The locale specifies the character encoding to use when writing to the InterSystems IRIS database. It also includes information necessary to handle character conversions to and from other character encodings.
- When you install an InterSystems IRIS server, the installer sets the default locale for that server.

  This cannot be changed after installation, but you can specify that a process uses a non-default locale, if wanted.

9.2 InterSystems IRIS Locales and National Language Support

An InterSystems IRIS *locale* is a set of metadata that defines storage and display conventions that apply to a specific country or geographic region. The locale definition includes the following:

- Number formats
- Date and time formats
- Currency symbols

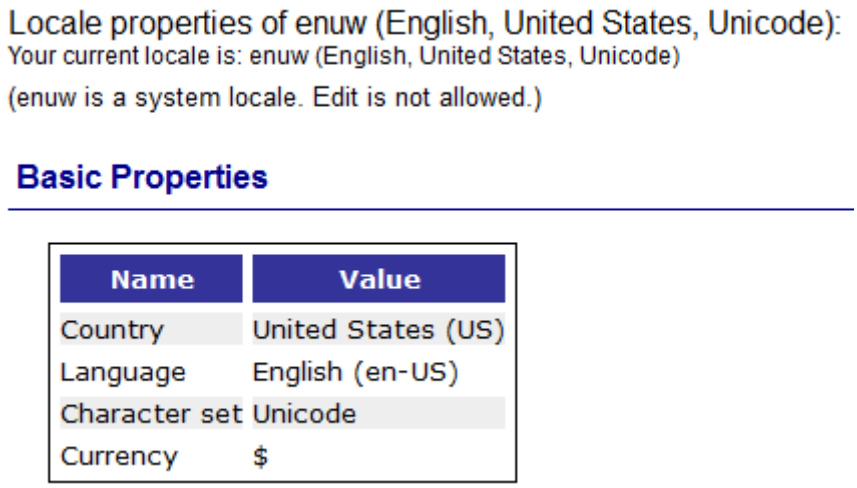
- The sort order of words
- The default character set (the character encoding of this locale), as defined by a standard (ISO, Unicode, or other).

  Note that InterSystems IRIS uses the phrases *character set* and *character encoding* as though they are synonymous, which is not strictly true in all cases.
- A set of *translation tables* (also called *I/O tables*) that convert characters to and from other supported character sets.

  The "translation table" for a given character set (for example, CP1250) is actually a pair of tables. One table specifies how to convert from the default character set to the foreign character set, and other specifies how to convert in the other direction. In InterSystems IRIS, the convention is to refer to this pair of tables as a single unit.

InterSystems IRIS uses the phrase *National Language Support* (NLS) to refer collectively to the locale definitions and to the tools that you use to view and extend them.

The Management Portal provides a page where you can see the default locale, view the details of any installed locale, and work with locales. The following shows an example:



You can also use this page to see the names of the available translation tables. These names are specific to InterSystems IRIS. (In some cases, it is necessary to know the names of these tables.)

For information on accessing and using this Management Portal page, see Using the NLS Pages of the Management Portal in the *System Administration Guide*.

InterSystems IRIS also provides a set of classes (in the %SYS.NLS and Config.NLS packages). See System Classes for National Language Support.

9.3 Default I/O Tables

External to the definition of any locale, a given InterSystems IRIS instance is configured to use specific translation tables, by default, for input/output activity. Specifically, it specifies the default translation tables to use in the following scenarios:

- When communicating with an InterSystems IRIS process
- When communicating with the InterSystems Terminal
- When reading from and writing to files
- When reading from and writing to TCP/IP devices

- When reading from and writing to strings sent to the operating system as parameters (such as file names and paths)
- When reading from and writing to devices such as printers

For example, when InterSystems IRIS needs to call an operating system function that receives a string as a parameter (such as a file name or path), it first passes the string through an NLS translation appropriately called `syscall`. The result of this translation is sent to the operating system.

To see the current defaults, use %SYS.NLS.Table; see the class reference for detail.

9.4 Files and Character Encoding

Whenever you read to or write from an entity external to the database, there is a possibility that the entity is using a different character set than InterSystems IRIS. The most common scenario is working with files.

At the lowest level, you use the Open command to open a file or other device. This command can accept a parameter that specifies the translation table to use when translating characters to or from that device. For details, see the *I/O Device Guide*. Then InterSystems IRIS uses that table to translate characters as needed.

Similarly, when you use the object-based file APIs, you specify the TranslateTable property of the file.

(Note that the production adapter classes instead provide properties to specify the foreign character set — to be used as the expected character encoding system of input data and the desired character encoding of output data. In this case, you specify a standard character set name, choosing from the set supported by InterSystems.)

9.5 Manually Translating Characters

InterSystems IRIS provides the $ZCONVERT function, which you can use to manually translate characters to or from another character set.

10

Server Configuration Options

There are a few configuration options for the server that can affect how you write your code.

Most of the configuration details are saved in a file called iris.cpf (the *configuration parameter file* or *CPF*).

10.1 Settings for InterSystems SQL

This section discusses some of the most important settings that affect the behavior of InterSystems SQL.

10.1.1 Adaptive Mode

This setting (which is on by default) ensures the best out-of-the-box performance for a wide set of use cases. Specifically, Adaptive Mode controls Runtime Plan Choice (RTPC), parallel processing, and automatically runs TUNE TABLE to optimize the efficiency of query execution. The individual features that Adaptive Mode governs cannot be controlled independently without turning Adaptive Mode off.

For details, see AdaptiveMode in the *Configuration Parameter File Reference*.

10.1.2 Retain Cached Query Source

This setting specifies whether to save the routine and INT code that InterSystems IRIS generates when you execute any InterSystems SQL except for embedded SQL. In all cases, the generated OBJ is kept. By default, the routine and INT code is not kept.

The query *results* are not stored in the cache.

For details, see SaveMAC in the *Configuration Parameter File Reference*.

10.1.3 Default Schema

This setting specifies the default schema name to use when creating or deleting tables that do not have a specified schema. It is also used for other DDL operations, such as creating or deleting a view, trigger, or stored procedure.

For details, see DefaultSchema in the *Configuration Parameter File Reference*.

For more information, see Schema Name.

10.1.4 Delimited Identifier Support

This setting controls how InterSystems SQL treats characters contained within a pair of double quotes.

If you enable support for delimited identifiers (the default), you can use double quotes around the names of fields, which enables you to refer to fields whose names are not regular identifiers. Such fields might, for example, use SQL reserved words as names.

If you disable support for delimited identifiers, characters within double quotes are treated as string literals, and it is not possible to refer to fields whose names are not regular identifiers.

You can set delimited identifier support system-wide using the SET OPTION command with the SUPPORT_DELIMITED_IDENTIFIERS keyword or by using the **$SYSTEM.SQL.Util.SetOption()** method `DelimitedIdentifiers` option. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

For more information, see Delimited Identifiers.

10.2 Use of IPv6 Addressing

InterSystems IRIS® data platform always accepts IPv4 addresses and DNS forms of addressing (host names, with or without domain qualifiers). You can configure InterSystems IRIS to also accept IPv6 addresses; see IPv6 Support in the *System Administration Guide*.

10.3 Configuring a Server Programmatically

You can programmatically change some of the operational parameters of InterSystems IRIS by invoking specific utilities; this is how you would likely change the configuration for your customers. For example:

- Config.Miscellaneous includes methods to set system-wide default and settings.
- %SYSTEM.Process includes methods to set environment values for the life of the current process.
- %SYSTEM.SQL includes methods for changing SQL settings.

For details, see the InterSystems Class Reference for these classes.

10.4 See Also

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

- *System Administration Guide* describes how to use most of the Management Portal.
- The InterSystems Class Reference provides details on all non-internal classes provided with InterSystems IRIS.
- *Configuration Parameter File Reference* contains reference information on the CPF.

11

Useful Skills to Learn

This page briefly describes some specific tasks that are useful for programmers to know how to perform. If you are familiar with how, when, and why to perform the tasks described here, you will be able to save yourself some time and effort.

11.1 Defining Databases

To create a local database:

1. Log in to the Management Portal.
2. Select **System Administration** > **Configuration** > **System Configurations** > **Local Databases**.
3. Select **Create New Database** to open the **Database Wizard**.
4. Enter the following information for the new database:
   - Enter a database name in the text box. Usually this is a short string containing alphanumeric characters; for rules, see Configuring Databases.
   - Enter a directory name or select **Browse** to select a database directory. If this is the first database you are creating, you must browse to the parent directory in which you want to create the database; if you created other databases, the default database directory is the parent directory of the last database you created.
5. Select **Finish**.

For additional options and information on creating remote databases, see Configuring System-Wide Settings.

11.2 Defining Namespaces

To create a namespace that uses local databases:

1. Log in to the Management Portal.
2. Select **System Administration** > **Configuration** > **System Configurations** > **Namespaces**.
3. Select **Create New Namespace**.
4. Enter a **Name for the namespace**. Usually this is a short string containing alphanumeric characters; for rules, see Configuring Namespaces.

5. For **Select an existing database for Globals**, select a database or select **Create New Database**.

   If you select **Create New Database**, the system prompts you with similar options as given in the create a database.

6. For **Select an existing database for Routines**, select a database or select **Create New Database**.

   If you select **Create New Database**, the system prompts you with similar options as when you create a database.

7. Select **Save**.

For additional options, see Configuring System-Wide Settings.

11.3 Mapping a Global

When you *map a global to database ABC*, you configure a given namespace so that InterSystems IRIS writes this global to and reads this global from the database ABC, which is not the default database for your namespace. When you define this *global mapping*, InterSystems IRIS does not move the global (if it already exists) to the designated database; instead the mapping instructs InterSystems IRIS where to read and write the global in the future.

To map a global:

1. If the global already exists, move it to the desired database. See Moving Data from One Database to Another.
2. Log in to the Management Portal.
3. Select **System Administration** > **Configuration** > **System Configurations** > **Namespaces**.
4. Select **Global Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Global Mapping**.
6. For **Global database location**, select the database that should store this global.
7. Enter the **Global name** (omitting the initial caret from the name). You can use the * character to choose multiple globals.

   The global does not have to exist when you map it (that is, it can be the name of a global you plan to create).

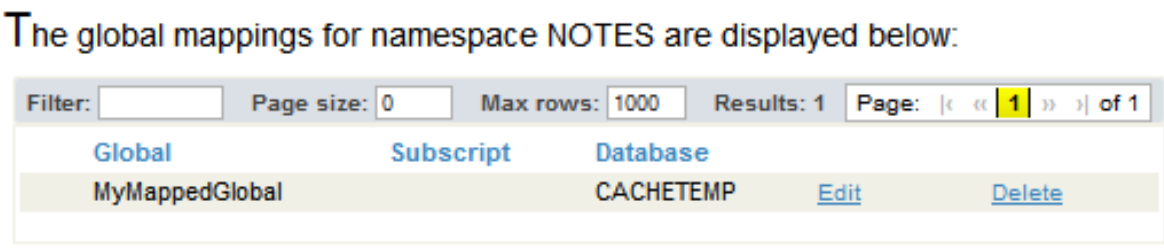
   Note: Typically you create mappings for data globals for persistent classes, because you want to store that data in non-default databases. Often you can guess the name of the data globals, but remember that InterSystems IRIS automatically uses a hashed form of the class name if the name is too long. It is worthwhile to check the storage definitions for those classes to make sure you have the exact names of the globals that they use. See Storage.

8. Select **OK**.
9. To save the mappings, select **Save Changes**.

For more information, see the *System Administration Guide*.

You can also define global mappings programmatically; see the Globals entry in the *InterSystems Programming Tools Index*.

The following shows an example global mapping, as seen in the Management Portal, which does not display the initial caret of global names:



This mapping means the following:

- Within the namespace `DEMONAMESPACE`, if you set values of nodes of the global `^MyTempGlobal`, you are writing data to the CACHETEMP database.

  This is true whether you set the nodes directly or indirectly (via object access or SQL).

- Within the namespace `DEMONAMESPACE`, if you retrieve values from the global `^MyTempGlobal`, you are reading data from the CACHETEMP database.

  This is true whether you retrieve the values nodes directly or indirectly (via object access or SQL).

11.4 Mapping a Routine

When you *map a routine to database ABC*, you configure a given namespace so that InterSystems IRIS finds this routine in the database ABC, which is not the default database for your namespace. When you define this *routine mapping*, InterSystems IRIS does not move the routine (if it already exists) to the designated database; instead the mapping instructs InterSystems IRIS where to find the routine in the future.

To map a routine:

1. If the routine already exists, copy it to the desired database by exporting it and importing it.
2. Log in to the Management Portal.
3. Select **System Administration** > **Configuration** > **System Configurations** > **Namespaces**.
4. Select **Routine Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Routine Mapping**.
6. For **Routine database location**, select the database that should store this routine.
7. Enter a value for **Routine name**. You can use the * character to choose multiple routines.

   Use the actual routine name; that is, do not include a caret (^) at the start.

   The routine does not have to exist when you map it (that is, it can be the name of a routine you plan to create).
8. Select the **Routine type**.
9. Select **OK**.
10. Select **OK**.
11. To save the mappings, select **Save Changes**.

For more information, see the *System Administration Guide*.

You can also define this kind of mapping programmatically. You can also define routine mappings programmatically; see the Routines entry in the *InterSystems Programming Tools Index*.

Important: When you map one or more routines, be sure to identify all the code and data needed by those routines, and ensure that all that code and data is available in all the target namespaces. The mapped routines could depend on the following items:

- Include files
- Other routines
- Classes
- Tables
- Globals

Use additional routine, package, and global mappings as needed to ensure that these items are available in the target namespaces.

11.5 Mapping a Package

When you *map a package to database ABC*, you configure a given namespace so that InterSystems IRIS finds the class definitions of this package in the database ABC, which is not the default database for your namespace. The mapping also applies to the generated routines associated with the class definitions; those routines are in the same package. This mapping does not affect the location of any stored data for persistent classes in these packages.

Also, when you define this *package mapping*, InterSystems IRIS does not move the package (if it already exists) to the designated database; instead the mapping instructs InterSystems IRIS where to find the package in the future.

To map a package:

1. If the package already exists, copy the package to the desired database by exporting and importing the classes.
2. Log in to the Management Portal.
3. Select **System Administration** > **Configuration** > **System Configurations** > **Namespaces**.
4. Select **Package Mappings** in the row for the namespace in which you want to define this mapping.
5. Select **New Package Mapping**.
6. For **Package database location**, select the database that should store this package.
7. Enter a value for **Package name**.

   The package does not have to exist when you map it (that is, it can be the name of a package you plan to create).
8. Select **OK**.
9. Select **OK**.
10. To save the mappings, select **Save Changes**.

For more information, see the *System Administration Guide*.

You can also define this kind of mapping programmatically. You can also define package mappings programmatically; see the Packages entry in the *InterSystems Programming Tools Index*.

Important: When you map a package, be sure to identify all the code and data needed by the classes in that package, and ensure that all that code and data is available in all the target namespaces. The mapped classes could depend on the following items:

- Include files
- Routines
- Other classes
- Tables
- Globals

Use additional routine, package, and global mappings as needed to ensure that these items are available in the target namespaces.

11.6 Generating Test Data

InterSystems IRIS includes a utility for creating pseudo-random test data for persistent classes. The creation of such data is known as *data population*, and the utility for doing this is known as the *populate utility*. This utility is especially helpful when testing how various parts of an application will function when working against a large set of data.

The populate utility consists of two classes: %Library.Populate and %Library.PopulateUtils. These classes provide methods that generate data of different typical forms. For example, one method generates random names:

ObjectScript

```
Write ##class(%Library.PopulateUtils).Name()
```

You can use the populate utility in two different ways.

11.6.1 Extending %Populate

In this approach, you do the following:

1. Add %Populate to the superclass list of your class.
2. Optionally specify a value for the *POPSPEC* parameter of each property in the class.

   For the value of the parameter, specify a method that returns a value suitable for use as a property value.

   For example:

   Class Member

   ```
   Property SSN As %String(POPSPEC = "##class(MyApp.Utils).MakeSSN()");
   ```

3. Write a utility method or routine that generates the data in the appropriate order: independent classes before dependent classes.

   In this code, to populate a class, execute the **Populate()** method of that class, which it inherits from the %Populate superclass.

   This method generates instances of your class and saves them by calling the **%Save()** method, which ensures that each property is validated before saving.

   For each property, this method generates a value as follows:

a. If the *POPSPEC* parameter is specified for that property, the system invokes that method and uses the value that it returns.

b. Otherwise, if the property name is a name such as `City`, `State`, `Name`, or other predefined values, the system invokes a suitable method for the value. These values are hardcoded.

c. Otherwise, the system generates a random string.

For details on how the %Populate class handles serial properties, collections, and so on, see Populate Utility.

4. Invoke your utility method from the Terminal or possibly from any applicable startup code.

This is the general approach used for Sample.Person in the SAMPLES database.

11.6.2 Using Methods of %Populate and %PopulateUtils

The %Populate and %PopulateUtils classes provide methods that generate values of specific forms. You can invoke these methods directly, in the following alternative approach to data population:

1. Write a utility method that generates the data in the appropriate order: independent classes before dependent classes.

   In this code, for each class, iterate a desired number of times. In each iteration:

   a. Create a new object.

   b. Set each property using a suitable random (or nearly random) value.

      To do so, use a method of %Populate or %PopulateUtils or use your own method.

   c. Save the object.

2. Invoke your utility method from the Terminal.

This is the approach used for the two DeepSee samples in the SAMPLES database, contained in the DeepSee and HoleFoods packages.

11.7 Removing Stored Data

During the development process, it may be necessary to delete all existing test data for a class and then regenerate it (for example, if you have deleted the storage definition).

Here are two quick ways to delete stored data for a class (additional techniques are possible):

- Call the following class method:

  ```
  ##class(%ExtentMgr.Util).DeleteExtent(classname)
  ```

  Where *classname* is the full package and class name.

- Delete the globals in which the data for the class and the indexes for the class are stored. You may be more comfortable doing this through the Management Portal:

  1. Select **System Explorer** > **Globals**.
  2. Select **Delete**.
  3. On the left, select the namespace in which you are working.
  4. On the right, select the check box next to the data global and the index global.

5. Select **Delete**.

   The system prompts to confirm that you want to delete these globals.

These options delete the data, but not the class definition. (Conversely, if you delete the class definition, that does not delete the data.)

11.8 Resetting Storage

Important: It is important to be able to reset storage during development, but you never do this on a live system.

The action of resetting storage for a class changes the way that the class accesses its stored data. If you have stored data for the class, and if you have removed, added, or changed property definitions, and you then reset storage, you might not be able to access the stored data correctly. So if you reset storage, you should *also* delete all existing data for the class and regenerate or reload it, as appropriate.

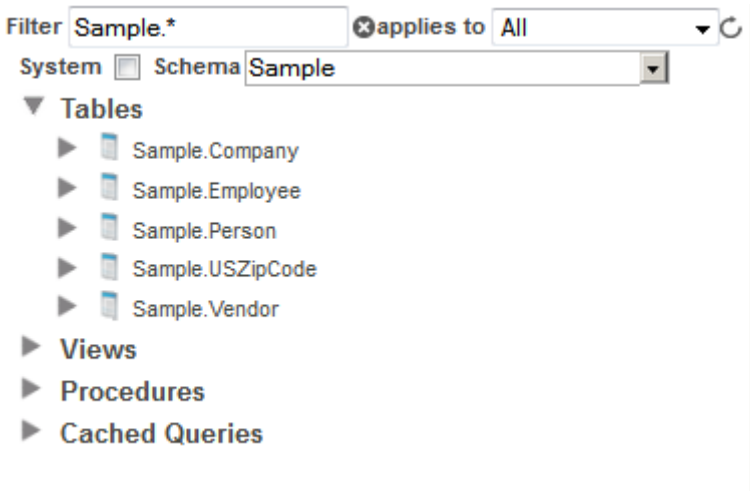
To reset storage for a class in your IDE:

1. Display the class.
2. Scroll to the end of the class definition.
3. Select the entire storage definition, starting with `<Storage name=` and ending with `</Storage>`. Delete the selection.
4. Save and recompile the class.

11.9 Browsing a Table

To browse a table, do the following in the Management Portal:

1. Select **System Explorer** > **SQL**.
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. Optionally select an SQL schema from the **Schema** drop-down list. This list includes all SQL schemas in this namespace. Each schema corresponds to a top-level class package.
4. Expand the **Tables** folder to see all the tables in this schema. For example:



5. Select the name of the table. The right area then displays information about the table.
6. Select **Open Table**.

   The system then displays the first 100 rows of this table. For example:

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

   Note the following points:

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

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

11.10 Executing an SQL Query

To run an SQL query, do the following in the Management Portal:

1. Select **System Explorer** > **SQL**.

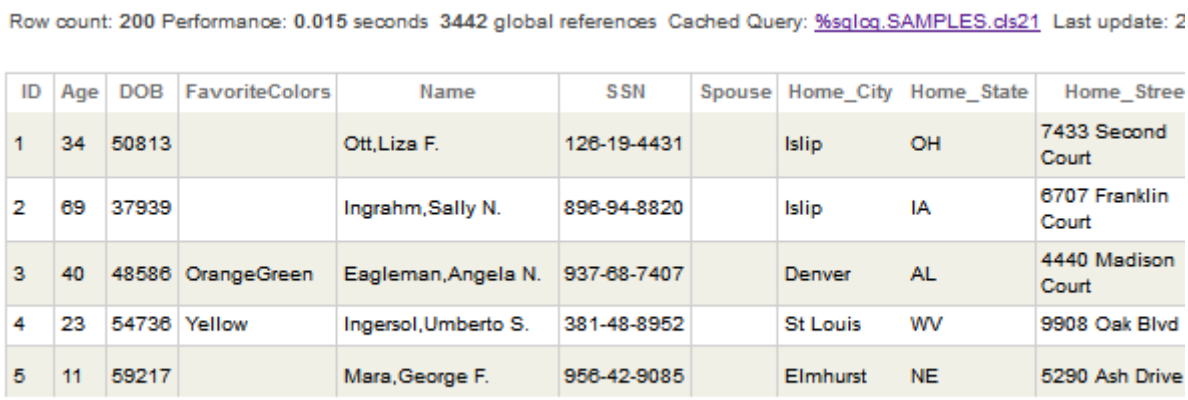
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. Select **Execute Query**.
4. Type an SQL query into the input box. For example:

   ```
   select * from sample.person
   ```

5. For the drop-down list, select **Display Mode**, **Logical Mode**, or **ODBC Mode**.

   This controls how the user interface displays the results.

6. Then select **Execute**. Then the Portal displays the results. For example:



Row count: 200 Performance: 0.015 seconds 3442 global references Cached Query: %sqlcq.SAMPLES.cls21 Last update: 2

ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Stree
1	34	50813		Ott,Liza F.	126-19-4431		Islip	OH	7433 Second Court
2	69	37939		Ingrahm,Sally N.	896-94-8820		Islip	IA	6707 Franklin Court
3	40	48586	OrangeGreen	Eagleman,Angela N.	937-68-7407		Denver	AL	4440 Madison Court
4	23	54736	Yellow	Ingersol,Umberto S.	381-48-8952		St Louis	WV	9908 Oak Blvd
5	11	59217		Mara,George F.	956-42-9085		Elmhurst	NE	5290 Ash Drive

11.11 Examining Object Properties

Sometimes the easiest way to see the value of a particular property is to open the object and write the property in the Terminal:

1. If the Terminal prompt is not the name of the namespace you want, then type the following and press return:

   ```
   ZN "namespace"
   ```

   Where *namespace* is the desired namespace.

2. Enter a command like the following to open an instance of this class:

   ```
   set object=##class(package.class).%OpenId(ID)
   ```

   Where *package.class* is the package and class, and *ID* is the ID of a stored object in the class.

3. Display the value of a property as follows:

   ```
   write object.propname
   ```

   Where *propname* is the property whose value you want to see.

11.12 Viewing Globals

To view globals in general, you can use the ObjectScript ZWRITE command or the **Globals** page in the Management Portal. If you are looking for the global that stores the data for a class, it is useful to first check the class definition to make sure you know the global to view.

1. If you are looking for the data global for a specific class and you are not sure which global stores the data for the class:

   a. In your IDE, display the class.

   b. Scroll to the end of the class definition.

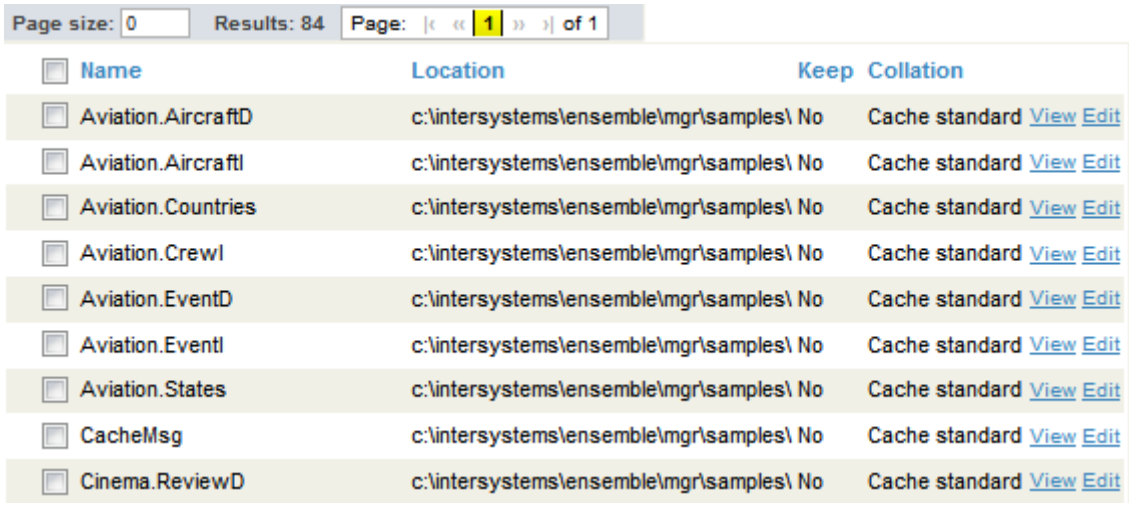
   c. Find the `<DefaultData>` element. The value between `<DefaultData>` and `</DefaultData>` is the name of the global that stores data for this class.

      InterSystems IRIS uses a simple naming convention to determine the names of these globals; see Globals Used by a Persistent Class. However, global names are limited to 31 characters (excluding the initial caret), so if the complete class name is long, the system automatically uses a hashed form of the class name instead.

2. In the Management Portal, select **System Explorer** > **Globals**.

3. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.

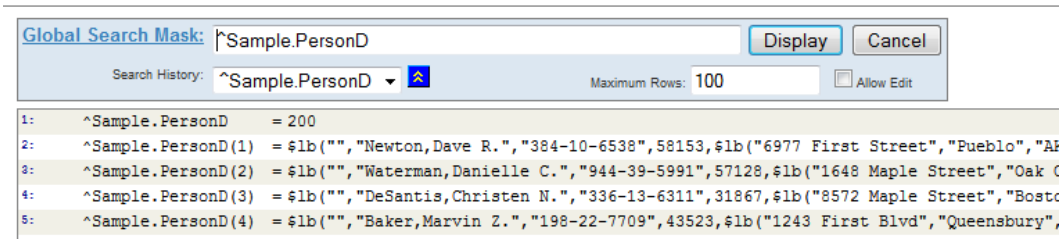
   The Portal displays a list of the globals available in this namespace (notice that this display omits the initial caret of each name). For example:



   Usually most non-system globals store data for persistent classes, which means that unless you display system globals, most globals will have familiar names.

4. Select **View** in the row for the global in which you are interested.

   The system then displays the first 100 nodes of this global. For example:



5. To restrict the display to the object in which you are interested, append (*ID*) to the end of the global name in the **Global Search Mask** field, using the ID of the object. For example:

   ```
   ^Sample.PersonD(45)
   ```

   Then press **Display**.

As noted earlier, you can also use the ZWRITE command, which you can abbreviate to ZW. Enter a command like the following in the Terminal:

```
zw ^Sample.PersonD(45)
```

11.13 Testing a Query and Viewing a Query Plan

In the Management Portal, you can test a query that your code will run. Here you can also view the query plan, which gives you information about how the Query Optimizer will execute the query. You can use this information to determine whether you should add indexes to the classes or write the query in a different way.

To view a query plan, do the following in the Management Portal:

1. Select **System Explorer** > **SQL**.
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. Select **Execute Query**.
4. Type an SQL query into the input box. For example:

   ```
   select * from sample.person
   ```

5. For the drop-down list, select **Display Mode**, **Logical Mode**, or **ODBC Mode**.

   This controls how the user interface displays the results.

6. To test the query, select **Execute**.
7. To see the query plan, select **Show Plan**.

11.14 Viewing the Query Cache

For InterSystems SQL (except when used as embedded SQL), the system generates reusable code to access the data and places this code in the *query cache*. (For embedded SQL, the system generates reusable code as well, but this is contained within the generated INT code.)

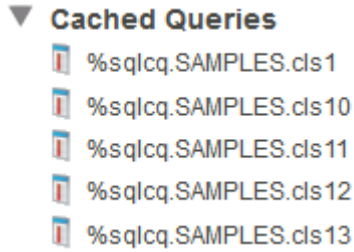
When you first execute an SQL statement, InterSystems IRIS optimizes the query and then generates and stores code that retrieves the data. It stores the code in the *query cache*, along with the optimized query text. Note that this cache is a cache of OBJ code, not of data.

Later when you execute an SQL statement, InterSystems IRIS optimizes it and then compares the text of that query to the items in the query cache. If InterSystems IRIS finds a stored query that matches the given one (apart from minor differences such as whitespace), it uses the code stored for that query.

The Management Portal groups the items in the query cache by schema. To view the query cache for a given schema, do the following in the Management Portal:

1. Select **System Explorer** > **SQL**.
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. Expand the **Cached Queries** folder.
4. Select the **Tables** ulink in the row for the schema.
5. At the top of the page, select **Cached Queries**.

   The Portal displays something like this:



   Each item in the list is OBJ code.

By default, InterSystems IRIS does not save the routine and INT code that it generates as a precursor to this OBJ code. You can force InterSystems IRIS to save this generated code as well. See Settings for InterSystems SQL.

You can purge cached queries (which forces InterSystems IRIS to regenerate this code). To purge cached queries, use **Actions** > **Purge Cached Queries**.

11.15 Building an Index

For InterSystems IRIS classes, indexes do not require any maintenance, with one exception: if you add an index after you already have stored records for the class, you must build the index.

To do so:

1. Select **System Explorer** > **SQL**.
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. In the left area, select the table.
4. Select **Actions** > **Rebuild Indices**.

11.16 Using the Tune Table Facility

When the Query Optimizer decides the most efficient way to execute a specific SQL query, it considers, among other factors, the following items:

- How many records are in the tables

- For the columns used by the query, how nearly unique those columns are

This information is available only if you have run the Tune Table facility with the given table or tables. This facility calculates this data and stores it with the storage definition for the class, as the `<ExtentSize>` value for the class and the `<Selectivity>` values for the stored properties.

To use the Tune Table facility:

1. Select **System Explorer** > **SQL**.
2. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
3. In the left area, select the table.
4. Select **Actions** > **Tune Table**.

For `<Selectivity>` values, it is not necessary to do this again unless the data changes in character. For `<ExtentSize>`, it is not important to have an exact number. This value is used to compare the relative costs of scanning over different tables; the most important thing is to make sure that the relative values of *ExtentSize* between tables are correct (that is, small tables should have a small value and large tables a large one).

11.17 Moving Data from One Database to Another

If you need to move data from one database to another, do the following:

1. Identify the globals that contain the data and its indexes.

   If you are not certain which globals a class uses, check its storage definition. See Storage.
2. Export those globals. To do so:
   a. In the Management Portal, select **System Explorer** > **Globals**.
   b. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
   c. Select the globals to export.
   d. Select **Export**.
   e. Specify the file into which you wish to export the globals. Either enter a file name (including its absolute or relative pathname) in the field or select **Browse** and navigate to the file.
   f. Select **Export**.

   The globals are exported to a file whose extensions is .gof.
3. Import those globals into the other namespace. To do so:
   a. In the Management Portal, select **System Explorer** > **Globals**.
   b. If needed, select the name of the current namespace displayed in the header area to select the namespace in which you are interested.
   c. Select **Import**.
   d. Specify the import file. Either enter the file name or select **Browse** and navigate to the file.

e. Select **Next** to view the contents of the file. The system displays a table of information about the globals in the specified file: the name of each global, whether or not it exists in the local namespace or database, and, if it does exist, when it was last modified.

f. Choose those globals to import using the check boxes in the table.

g. Select **Import**.

4. Go back to the first database and delete the globals, as described in Removing Stored Data.

A

What's That?

As you read existing ObjectScript code, you may encounter unfamiliar syntax forms. This page shows syntax forms in different groups, and it explains what they are and where to find more information.

This page does not list single characters that are obviously operators or that are obviously arguments to functions or commands.

A.1 Non-Alphanumeric Characters in the Middle of "Words"

This section lists forms that look like words with non-alphanumeric characters in them. Many of these are obvious, because the operators are familiar. For example:

```
x>5
```

The less obvious forms are these:

`abc^def`

`def` is a routine, and `abc` is a label within that routine. `abc^def` is a subroutine.

Variation for `abc`:

- `%abc`

Some variations for `def`:

- `%def`
- `def.ghi`
- `%def.ghi`
- `def(xxx)`
- `%def(xxx)`
- `def.ghi(xxx)`
- `%def.ghi(xxx)`

`xxx` is an optional, comma-separated list of arguments.

A label can start with a percent sign but is purely alphanumeric after that.

A routine name can start with a percent sign and can include one or more periods. The caret is not part of its name. (In casual usage, however, it is very common to refer to a routine as if its name included an initial caret. Thus you may see comments about the `^def` routine. Usually you can tell from context whether the reference is to a global or to a routine.)

`i%abcdef`

This is an *instance variable*, which you can use to get or to set the value of the `abcdef` property of an object. See Object-specific ObjectScript Features.

This syntax can be used only in an instance method. `abcdef` is a property in the same class or in a superclass.

`abc->def`

Variations:

- `abc->def->ghi` and so on

This syntax is possible only within InterSystems SQL statements. It is an example of InterSystems IRIS *arrow syntax* and it specifies an implicit left outer join. `abc` is an object-valued field in the class that you are querying, and `def` is a field in the child class.

`abc->def` is analogous to dot syntax (`abc.def`), which you cannot use in InterSystems SQL.

For information on InterSystems IRIS arrow syntax, see Implicit Joins (Arrow Syntax).

`abc?def`

Variation:

- `"abc"?def`

A question mark is the pattern match operator. In the first form, this expression tests whether the value in the variable `abc` matches the pattern specified in `def`. In the second form, `"abc"` is a string literal that is being tested.

Note that both the string literal `"abc"` and the argument `def` can include characters other than letters.

`"abc"["def"`

Variations:

- `abc[def`
- `abc["def"`
- `"abc"[def`

A left bracket (`[`) is the binary contains operator. In the first form, this expression tests whether the string literal `"abc"` contains the string literal `"def"`. In later forms, `abc` and `def` are variables that are being tested.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

`"abc"]"def"`

Variations:

- `abc]def`
- `abc]"def"`
- `"abc"]def`

A right bracket (`]`) is the binary follows operator. In the first form, this expression tests whether the string literal `"abc"` comes after the string literal `"def"`, in ASCII collating sequence. In later forms, `abc` and `def` are variables that are being tested.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

`"abc"]]"def"`

Variations:

- `abc]]def`
- `abc]]"def"`
- `"abc"]]def`

Two right brackets together (`]]`) are the binary sorts after operator. In the first form, this expression tests whether the string literal `"abc"` sorts after the string literal `"def"`, in numeric subscript collation sequence. In later forms, `abc` and `def` are variables that are being tested.

Note that both the string literals `"abc"` and `"def"` can include characters other than letters.

A.2 . (One Period)

period within an argument list

Variations:

- `abc.def(.ghi)`
- `abc(.xyz)`

When you call a method or routine, you can pass an argument by reference or as output. To do so, place a period before the argument.

period at the start of a line

An older form of the Do command uses a period prefix to group lines of code together into a code block. This older Do command is not for use with InterSystems IRIS.

A.3 .. (Two Periods)

In every case, two periods together are the start of a reference from within a class member to another class member.

`..abcdef`

This syntax can be used only in an instance method (not in routines or class methods). `abcdef` is a property in the same class.

`..abcdef(xxx)`

This syntax can be used only in a method (not in routines). `abcdef()` is another method in the same class, and `xxx` is an optional comma-separated list of arguments.

..#abcdef

This syntax can be used only in a method (not in routines). abcdef is a parameter in this class.

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

Remember that the pound sign is *not* part of the parameter name.

A.4 ... (Three Periods)

In the argument list of a method or a procedure, the last argument can be followed by three periods.

abcdef...

abcdef is an argument of the method or procedure and typically has a generic name such as arguments. The three periods indicate that additional arguments are accepted at this position in the argument list. See Specifying a Variable Number of Arguments and Variable Number of Parameters. When calling the method or procedure, do not include the three periods; simply include the arguments as needed, in the order needed, at this position in the argument list.

A.5 # (Pound Sign)

This section lists forms that start with a pound sign.

#abcdef

In most cases, #abcdef is a preprocessor directive. InterSystems IRIS provides a set of preprocessor directives. Their names start with either one or two pound signs. Here are some common examples:

- **#define** defines a macro (possibly with arguments)
- **#def1arg** defines a macro that has one argument that includes commas
- **#sqlcompile mode** specifies the compilation mode for any subsequent embedded SQL statements

For reference information and other directives, see ObjectScript Macros and the Macro Preprocessor.

Less commonly, the form #abcdef is an argument used with specific commands (such as READ and WRITE), special variables, or routines. For details, consult the reference information for the command, variable, or routine that uses this argument.

##abcdef

##abcdef is a preprocessor directive. See the comments for #abcdef.

##class(abc.def).ghi(xxx)

Variation:

- ##class(def).ghi(xxx)

abc.def is a package and class name, ghi is a class method in that class, and xxx is an optional comma-separated list of arguments.

If the package is omitted, the class `def` is in the same package as the class that contains this reference.

`##super()`

Variations:

- `##super(abcdef)`

This syntax can be used only in a method. It invokes the overridden method of the superclass, from within the current method of the same name in the current class. `abcdef` is a comma-separated list of arguments for the method. See Object-specific ObjectScript Features in Defining and Using Classes.

A.6 Dollar Sign ($)

This section lists forms that start with a dollar sign.

`$abcdef`

Usually, `$abcdef` is a special variable. See ObjectScript Special Variables in the *ObjectScript Reference*.

`$abcdef` could also be a custom special variable. See Extending ObjectScript with %ZLang.

`$abcdef(xxx)`

Usually, `$abcdef()` is a system function, and `xxx` is an optional comma-separated list of arguments. For reference information, see the *ObjectScript Reference*.

`$abcdef()` could also be a custom function. See Extending ObjectScript with %ZLang.

`$abc.def.ghi(xxx)`

In this form, `$abc` is `$SYSTEM` (in any case), `def` is the name of class in the `%SYSTEM` package, `ghi` is the name of a method in that class, and `xxx` is an optional comma-separated list of arguments for that method.

The **$SYSTEM** special variable is an alias for the `%SYSTEM` package, to provide language-independent access to methods in classes of that package. For example: **$SYSTEM.SQL.DATEDIFF**

For information on the methods in this class, see the InterSystems Class Reference.

`$$abc`

Variation:

- `$$abc(xxx)`

`abc` is a subroutine defined within the routine or the method that contains this reference. This syntax invokes the subroutine `abc` and gets its return value. See User-defined Code in Using ObjectScript.

`$$abc^def`

Variations:

- `$$abc^def(xxx)`
- `$$abc^def.ghi`
- `$$abc^def.ghi(xxx)`

This syntax invokes the subroutine `abc` and gets its return value. The part after the caret is the name of the routine that contains this subroutine. See User-defined Code in Using ObjectScript.

`$$$abcdef`

`abcdef` is a macro; note that the dollar signs are not part of its name (and are thus not seen in the macro definition).

Some of the macros supplied by InterSystems IRIS are documented in System-Supplied Macro Reference.

In casual usage, it is common to refer to a macro as if its name included the dollar signs. Thus you may see comments about the `$$$abcdef` macro.

A.7 Percent Sign (%)

By convention, most packages, classes, and methods in InterSystems IRIS system classes start with a percent character. From the context, it should be clear whether the element you are examining is one of these. Otherwise, the possibilities are as follows:

`%abcdef`

`%abcdef` is one of the following:

- A local variable, including possibly a local variable set by InterSystems IRIS.
- A routine.

  Variation:

  – `%abcdef.ghijkl`

- An embedded SQL variable (these are *%msg*, *%ok*, *%ROWCOUNT*, and *%ROWID*).

  For information, see System Variables.

- An InterSystems SQL command, function, or predicate condition (for example, %STARTSWITH and %SQLUPPER).

  Variation:

  – `%abcdef(xxx)`

  For information, see the *InterSystems SQL Reference*.

`%%abcdef`

`%abcdef` is %%CLASSNAME, %%CLASSNAMEQ, %%ID, or %%TABLENAME. These are pseudo-field keywords. For details, see the *InterSystems SQL Reference*.

A.8 Caret (^)

This section lists forms that start with a caret, from more common to less common.

`^abcdef`

Variation:

- `^%abcdef`

There are three possibilities:

- `^abcdef` or `^%abcdef` is a global.
- `^abcdef` or `^%abcdef` is an argument of the **LOCK** command. In this case, `^abcdef` or `^%abcdef` is a lock name and is held in the lock table (in memory).
- `abcdef` or `%abcdef` is a routine. The caret is not part of the name, but rather part of the syntax to call the routine.

In casual usage, it is very common to refer to a routine as if its name included an initial caret. Thus you may see comments about the `^abcdef` routine. Usually you can tell from context whether the reference is to a global or to a routine. Lock names appear only after the **LOCK** command; they cannot be used in any other context.

`^$abcdef`

Variation:

- `^$|"ghijkl"|abcdef`

Each of these is a structured system variable, which provides information about globals, jobs, locks, or routines.

`$abcdef` is `$GLOBAL`, `$JOB`, `$LOCK`, or `$ROUTINE`.

`ghijkl` is a namespace name.

InterSystems IRIS stores information in the following system variables:

- ^$GLOBAL
- ^$JOB
- ^$LOCK
- ^$ROUTINE

See the *ObjectScript Reference*.

`^||abcdef`

Variations:

- `^|"^"|abcdef`
- `^["^"]abcdef`
- `^["^",""]abcdef`

Each of these is a *process-private global*, a mechanism for temporary storage of large data values. InterSystems IRIS uses some internally but does not present any for public use. You can define and use your own process-private globals. See Variables.

`^|XXX|abcdef`

Some variations:

- `^|XXX|%abcdef`
- `^[XXX]abcdef`
- `^[XXX]%abcdef`

Each of these is an *extended reference* — a reference to a global or a routine in another namespace. The possibilities are as follows:

- `^abcdef` or `^%abcdef` is a global in the other namespace.
- `abcdef` or `%abcdef` is a routine in the other namespace.

The `XXX` component indicates the namespace. This is either a quoted string or an unquoted string. See "Extended References" in Syntax Rules.

`^abc^def`

This is an implied namespace. See ZNSPACE in the *ObjectScript Reference*.

`^^abcdef`

This is an implied namespace. See the ZNSPACE entry in the *ObjectScript Reference*.

A.9 Other Forms

`+abcdef`

Some variations:

- `+^abcdef`
- `+"abcdef"`

Each of these expressions returns a number. In the first version, `abcdef` is the name of a local variable. If the contents of this variable do not start with a numeric character, the expression returns 0. If the contents do start with a numeric character, the expression returns that numeric character and all numeric characters after it, until the first nonnumeric character. For a demonstration, run the following example:

ObjectScript

```
write +"123abc456"
```

See String Relational Operators.

`{"abc":(def),"abc":(def),"abc":(def)}`

This syntax is a JSON object literal and it returns an instance of %DynamicObject. `"abc"` is the name of a property, and `def` is the value of the property. For details, see *Using JSON*.

`{abcdef}`

This syntax is possible where InterSystems SQL uses ObjectScript. `abcdef` is the name of a field. See Referring to Fields from ObjectScript in Defining and Using Classes.

`{%%CLASSNAME}`

This syntax can be used within trigger code and is replaced at class compilation time.

Others:

- `{%%CLASSNAMEQ}`
- `{%%ID}`

- {%%TABLENAME}

These items are not case-sensitive. See CREATE TRIGGER in the *InterSystems SQL Reference*.

&sql(xxx)

This is embedded SQL and can be used anywhere ObjectScript is used. xxx is one SQL statement. See Using Embedded SQL.

[abcdef,abcdef,abcdef]

This syntax is a JSON array literal and it returns an instance of %DynamicArray. abcdef is an item in the array. For details, see *Using JSON*.

***abcdef**

Special syntax used by the following functions and commands:

- $ZSEARCH
- $EXTRACT
- WRITE
- $ZTRAP
- $ZERROR

See these items in the *ObjectScript Reference*.

?abcdef

The question mark is the pattern match operator and abcdef is the comparison pattern.

@abcdef

The at sign is the indirection operator.

A.10 See Also

- Symbols Used in ObjectScript
- Abbreviations Used in ObjectScript