InterSystems™
IRIS Data Platform

Orientation Guide for Server-Side Programming

Version 2025.1
2025-06-03

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

Table of Contents

1

Introduction to InterSystems IRIS Programming

This page provides a high-level introduction to applications in InterSystems IRIS® data platform, with emphasis on server-side code.

1.1 Introduction

InterSystems IRIS is a high-performance multi-model data platform with an implementation of SQL, known as InterSystems SQL, as well as built-in support for Python, and a general-purpose programming language called ObjectScript. InterSystems IRIS also provides support for class definitions; the procedural parts of a class can be implemented in Python or in ObjectScript. A set of system classes enable you to model objects and persistent objects (tables). Additional system classes provide a wide set of APIs.

InterSystems IRIS programming has the following features:

- InterSystems IRIS supports multiple processes and provides concurrency control. Each process has direct, efficient access to the data.
- Database storage is available directly via SQL, Python, and ObjectScript.
- In all cases, stored data is ultimately contained in structures known as globals.

You can execute SQL, Python, and ObjectScript commands in a command-line shell, which is a useful tool to complement your IDE. You can also execute SQL in the Management Portal.

InterSystems IRIS supports Unicode data and localization.

1.2 Components of an InterSystems IRIS Application

From the point of view of an application developer, an InterSystems IRIS application consists of some or all the following elements on an InterSystems IRIS server:

- An application definition, which controls access to the code.

  This controls things such as the allowed authentication mechanisms and the InterSystems IRIS namespace to start running the code in.

- Any mix of InterSystems IRIS class definitions, routines, and include files that are available in that namespace. Classes and routines can be used interchangeably and can call each other. Include files provide central definitions for macros used within either classes or routines.
- One or more namespaces, including the namespace in which the application starts running.
- One or more databases, containing the data and code for the application.
- External files as needed.

In InterSystems IRIS, you can access data directly from SQL, or classes, or routines, which means that you have the flexibility to store and access data exactly how you want.

1.3 Classes

InterSystems IRIS supports classes and object-oriented programming. 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 topics provide more information on classes in InterSystems IRIS and the unique capabilities of persistent classes in InterSystems IRIS.

1.4 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"))
    }
```

1.5 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
}
```

1.6 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.

1.7 InterSystems SQL

InterSystems IRIS provides an implementation of SQL, known as InterSystems SQL. You can use InterSystems SQL within methods and within routines. Also, as noted previously, you can use SQL DDL commands to create tables (persistent classes).

1.7.1 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.

1.7.2 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.

1.8 Macros and Include Files

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
```

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 details, see Using Macros and Include Files.

The phrase *include file* is used for historical reasons. In InterSystems IRIS, an include file is not actually a separate standalone file in the operating system; instead it is a unit of code stored within an InterSystems IRIS database.

1.9 Location of Code

In InterSystems IRIS, class definitions, routines, and include files are stored within an InterSystems IRIS database. Your IDE provides options for editing this code and synchronizing with your choice of source control system.

1.10 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, and you can see it within your IDE.

- 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 final 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.

The intermediate code and the final code are contained in the same database that contains the code from which it was generated.

2 Introduction to Classes

This page introduces classes in InterSystems IRIS® data platform and shows how they can contain methods written in Python or ObjectScript. Class definitions are not formally part of either language; instead, the syntax used to define classes is called Class Definition Language.

Other method implementation languages are supported as well, primarily for migration purposes.

2.1 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.

2.2 Basic Contents of a Class Definition

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

- Parameters — A parameter defines a constant value for use by this class. The value is set at compilation time.
- Properties — A property contains data for an instance of the class.
- Methods — There are two kinds of methods: instance methods and class methods (called static methods in other languages).
- 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. See Defining and Using Class Queries.
- XData blocks — An XData block is a unit of data (XML, JSON, or YAML), for use by the class. See Defining and Using XData Blocks.
- Other kinds of class members that are relevant only for persistent classes.

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 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/Python

```
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 = python ]
 {
    import iris
    x = iris.MyApp.Main.SampleClass._GetParameter("CONSTANTMESSAGE")
    return x
 }

Method WriteIt() [ ServerOnly = 1, Language = python ]
{
    import iris
    count = self.MessageCount
    print()
    for i in range(count):
        print(iris.cls(__name__)._GetParameter("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.

  To access class parameters via Python, use the built-in method **%GetParameter()** to return the value of the parameter. However, in Python the percent sign character is not permitted in identifier names, so you need to substitute an underscore, instead.

- To refer to the current class in ObjectScript, use a dot (`.`).

- To refer to the current class in Python, you can use `self`, `iris.Package.Class`, or `iris.cls(__name__)`, depending on the context. The example shows all three syntax forms. Note that `iris.cls(__name__)` has two underscores before and after `name`. (For more details, see References to Other Class Members.)

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

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

Note: Effective with InterSystems IRIS 2024.2, an optional shorter syntax for referring to an InterSystems IRIS class from Embedded Python has been introduced. Either the new form or the traditional form are permitted.

2.3 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.

2.4 Properties

Object classes can have 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()**.

2.5 Methods

There are two kinds of methods: instance methods and class methods (called static methods in other languages). Instance methods are available only in object classes.

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 default language for the class.

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

2.5.1 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.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.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()
```

2.5.2 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.Package.Class.MethodName(arguments)
  ```

  For example:

  ObjectScript

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

  Python

  ```
  x = iris.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.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.

2.5.3 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)
```

2.6 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.

2.6.1 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
```

2.6.2 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.

2.6.3 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])
}
```

2.6.4 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.

2.7 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, 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.

When calling class methods from Python, you can omit the package (or higher level packages) if using the **iris.cls()** method, as in `s = iris.cls('%String')`. Use the longer form if using iris.Package.Class, as in `s = iris._Library.String`.

2.8 See Also

- Introduction to Objects
- Defining Classes
- Defining and Calling Methods
- Class Definition Reference
- Introduction to ObjectScript

3 Introduction to Objects

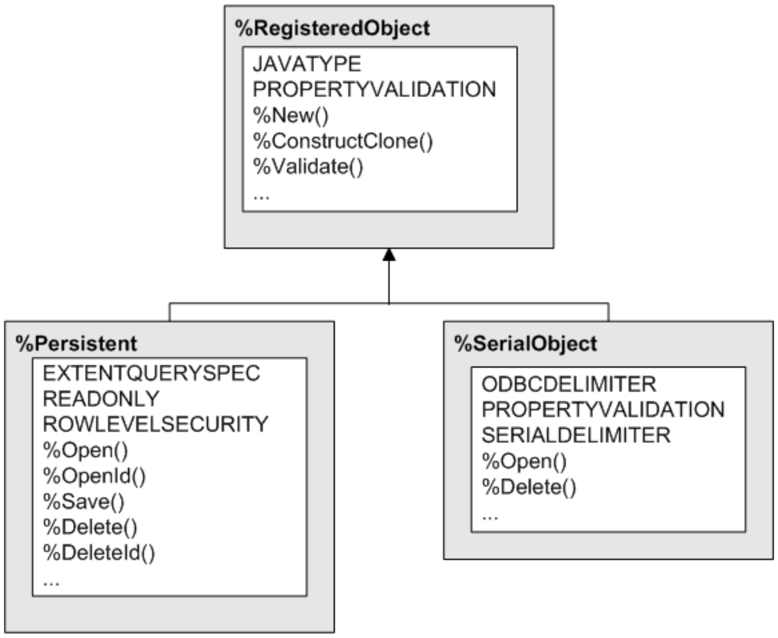
This page discusses objects in InterSystems IRIS® data platform.

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.

3.1 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.

3.2 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.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.Sample.Address._New()
myaddress.City = "Louisville"
myaddress.Street = "15 Winding Way"
myaddress.State = "Georgia"

myperson = iris.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.

3.3 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>write myobj
  3@Sample.Person
  ```

  #### Python Shell

  ```
  >>> myobj = iris.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.

3.4 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.

3.4.1 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()**.

3.4.2 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()
```

3.5 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.

3.5.1 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._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._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.

3.5.2 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.

3.6 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.

3.7 See Also

- Defining and Using Classes
- Class Definition Reference
- Downloading Samples for Use with InterSystems IRIS

4 Persistent Objects and InterSystems 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*.

4.1 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.)

4.2 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*.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.3 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.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.

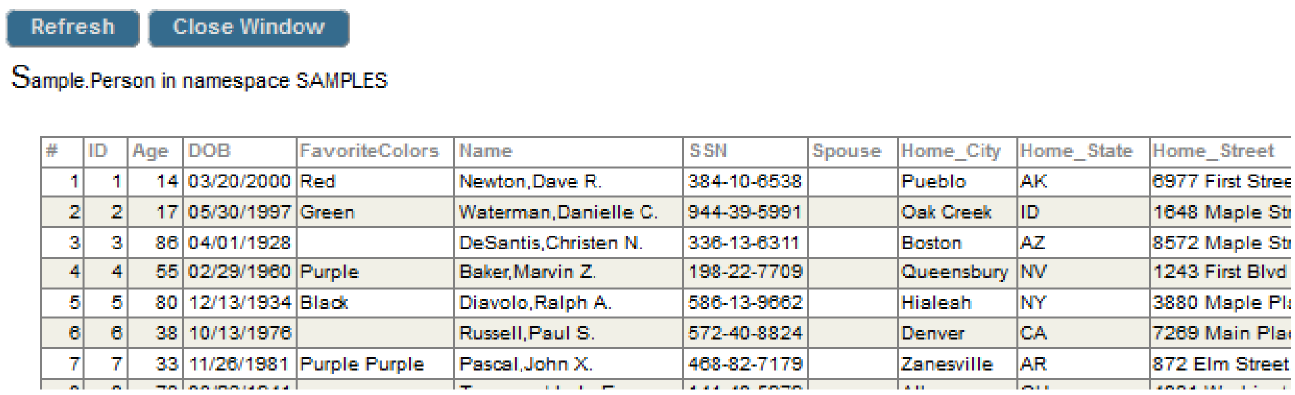
Note: You cannot define relationships, foreign keys, or indexes in Python.

4.4 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.

4.4.1 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:



Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First 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.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.

4.4.2 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.

4.4.3 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.

4.5 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.

4.5.1 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.

4.5.2 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

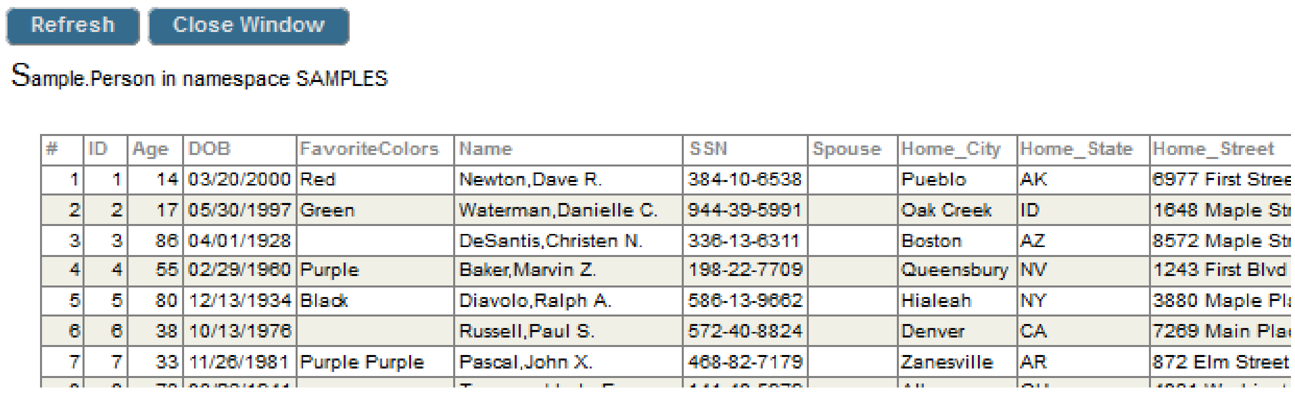
```
SAMPLES>set person=##class(Sample.Person).%OpenId(2)

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

Python Shell

```
>>> person = iris.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:



Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First 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**.

4.6 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.

4.6.1 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>
```

4.6.2 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*.

4.6.3 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.

4.7 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.

4.8 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.

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

Python Shell

```
>>> person = iris.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:

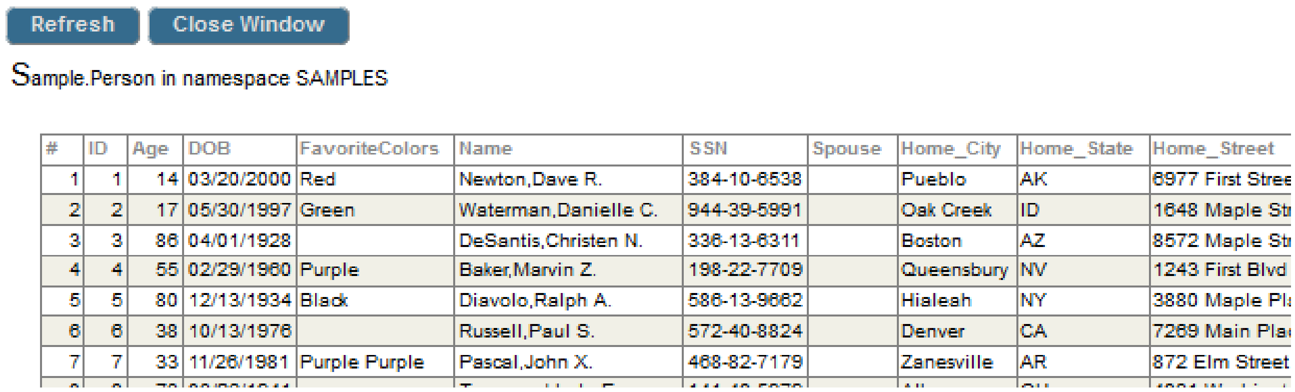
ObjectScript Shell

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

Python Shell

```
>>> print(iris._Library.Date.LogicalToDisplay(person.DOB))
03/20/2000
```

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

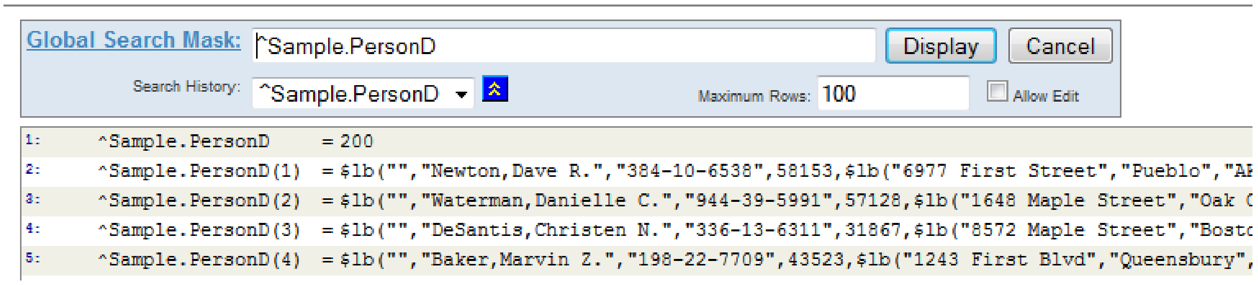


Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First 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.	586-13-9862		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:



```
1:   ^Sample.PersonD     = 200
2:   ^Sample.PersonD(1)  = $lb("","Newton,Dave R.","384-10-6538",58153,$lb("6977 First Street","Pueblo","AK
3:   ^Sample.PersonD(2)  = $lb("","Waterman,Danielle C.","944-39-5991",57128,$lb("1648 Maple Street","Oak C
4:   ^Sample.PersonD(3)  = $lb("","DeSantis,Christen N.","336-13-6311",31867,$lb("8572 Maple Street","Bosto
5:   ^Sample.PersonD(4)  = $lb("","Baker,Marvin Z.","198-22-7709",43523,$lb("1243 First Blvd","Queensbury",
```

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

4.10 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.

4.11 See Also

- Defining and Using Classes
- Class Definition Reference
- Using InterSystems SQL
- InterSystems SQL Reference
- Using Globals

Also the InterSystems Class Reference has information on all non-internal classes provided by InterSystems IRIS.

5 Namespaces and Databases

This page describes how InterSystems IRIS® data platform organizes data and code.

5.1 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.

5.1.1 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.

5.2 Database Basics

An InterSystems IRIS *database* is an IRIS.DAT file. You create a database via configuration tools such as the Management Portal. Or if you have an existing InterSystems IRIS database, you can configure InterSystems IRIS to become aware of it.

5.2.1 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.

  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 databases follow this convention.

Additional options include the following:

- Default directory to use for file streams used by this database.

  This is important because your users will need write access to this directory; if not, your code will not be able to create file streams.

- Collation of new globals.
- Initial size and other physical characteristics.
- Option to enable or disable *journaling*. Journaling tracks changes made to a 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.

5.2.2 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.

5.2.3 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.

5.3 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.

CAUTION: InterSystems does not support moving the IRISLIB database.

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.

IRISMETRICS

Read/write database that is meant to contain interoperability metrics data.

Note: No customer application should directly interact with the IRISMETRICS database. This database is purely for internal use by InterSystems IRIS.

IRISSYS (the *system manager's database*)

Read/write database that 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.

5.4 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.

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.

5.5 %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.

5.6 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.
- Specific globals as follows:
  - Any global whose name starts with a caret and a percent sign (`^%`). These globals are generally referred to as *global percent variables* or *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 in this namespace.

    Note that InterSystems IRIS provides special treatment for 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; see Temporary Globals and the IRISTEMP Database.

  See Variables for more information on variable names, their scope, and special characteristics.
- 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 references, your code can access globals and routines that are defined in other namespaces; see Extended References.

The InterSystems IRIS security model controls which data and which code any user can access.

5.6.1 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 Variable Names to Avoid.

5.7 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.

5.8 See Also

- Using Globals
- Using cvendian to Convert Between Big-endian and Little-endian Systems
- High Availability Guide
- Scalability Guide

6

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.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

6.5 See Also

- About InterSystems Security

7 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.

7.1 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.

7.2 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.)

7.3 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.

7.4 See Also

- Defining Applications (includes information on programmatic role escalation)
- Custom Web APIs and Applications

8
Localization in InterSystems IRIS

This page provides an overview of InterSystems IRIS® data platform support for localization.

8.1 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.

8.2 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

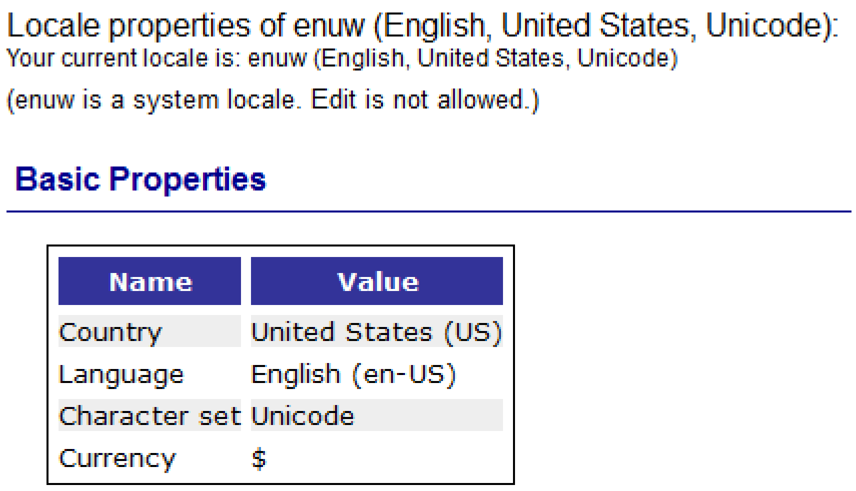
- The sort order of words
- The default character set (the character encoding of this locale), as defined by a standard (ISO, Unicode, or other).

  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:


Locale properties of enuw (English, United States, Unicode):
Your current locale is: enuw (English, United States, Unicode)
(enuw is a system locale. Edit is not allowed.)

Basic Properties

Name	Value
Country	United States (US)
Language	English (en-US)
Character set	Unicode
Currency	$

You can also use this page to see the names of the available translation tables. These names are specific to 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.

InterSystems IRIS also provides a set of classes (in the %SYS.NLS and Config.NLS packages). See System Classes for National Language Support.

8.3 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.

8.4 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 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.)

8.5 Manually Translating Characters

InterSystems IRIS provides the $ZCONVERT function, which you can use to manually translate characters to or from another character set.

8.6 See Also

- String Localization and Message Dictionaries
- Translation Tables
- System Classes for National Language Support
- $ZCONVERT

9 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*).

9.1 Settings for InterSystems SQL

This section discusses some of the most important settings that affect the behavior of InterSystems SQL.

9.1.1 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*.

9.1.2 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*.

9.1.3 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.

9.1.4 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.

9.2 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.

9.3 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.

9.4 See Also

- System Administration Guide
- Configuration Parameter File Reference

10 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.

10.1 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.

10.2 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.

10.3 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).

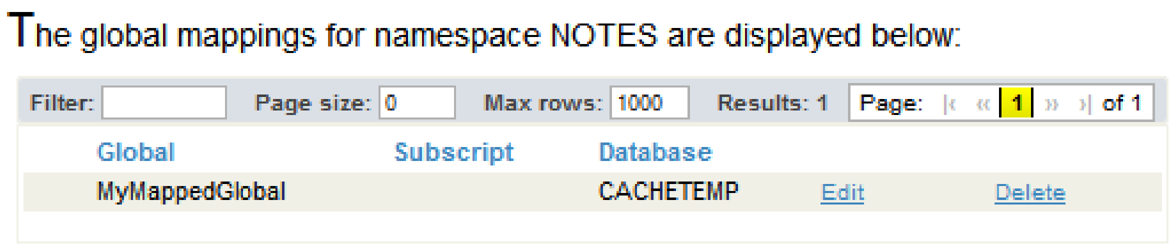
   Note: Typically you create mappings for data globals for persistent classes, because you want to store that data in non-default databases. Often you can guess the name of the data globals, but remember that InterSystems IRIS automatically uses a hashed form of the class name if the name is too long. 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:


The global mappings for namespace NOTES are displayed below:

Filter: Page size: 0 Max rows: 1000 Results: 1 Page: 1 of 1

Global	Subscript	Database		
MyMappedGlobal		CACHETEMP	Edit	Delete

This mapping means the following:

- Within the namespace DEMONAMESPACE, if you set values of nodes of the global `^MyTempGlobal`, you are writing data to the CACHETEMP database.

  This is true whether you set the nodes directly or indirectly (via object access or SQL).

- Within the namespace DEMONAMESPACE, if you retrieve values from the global `^MyTempGlobal`, you are reading data from the CACHETEMP database.

  This is true whether you retrieve the values nodes directly or indirectly (via object access or SQL).

10.4 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.

10.5 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.

10.6 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.

10.6.1 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.

10.6.2 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.

10.7 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.)

10.8 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.

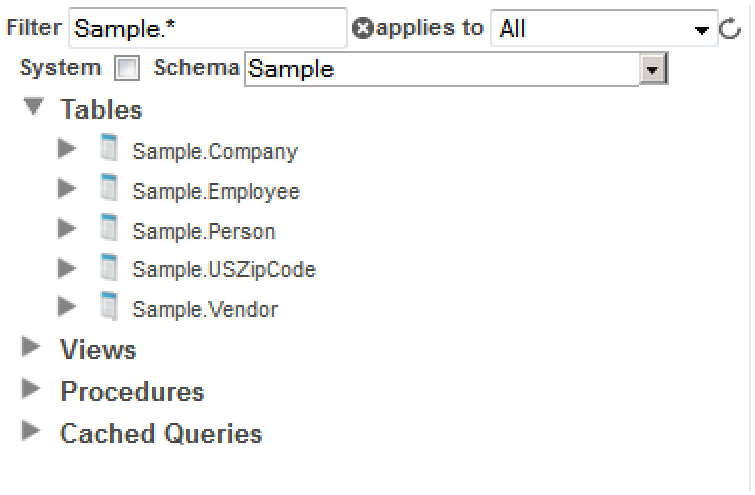
To reset storage for a class in your IDE:

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.

10.9 Browsing a Table

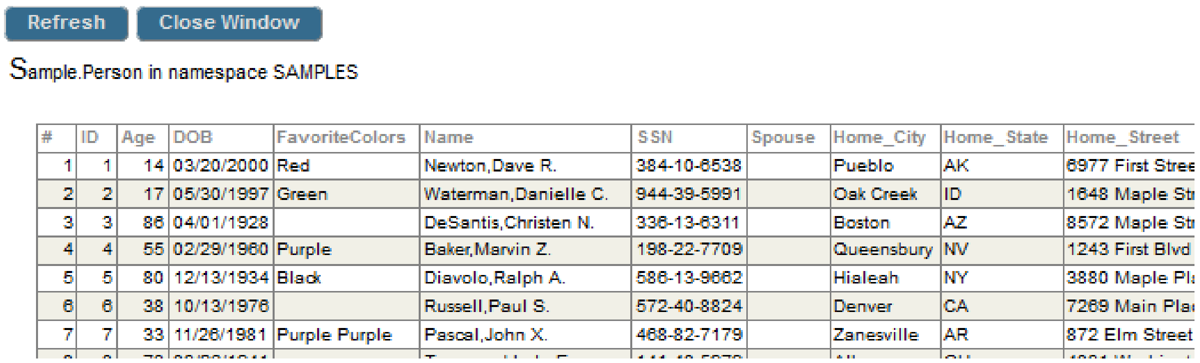
To browse a table, do the following in the Management Portal:

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:



Sample.Person in namespace SAMPLES

#	ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street
1	1	14	03/20/2000	Red	Newton,Dave R.	384-10-6538		Pueblo	AK	6977 First Stree
2	2	17	05/30/1997	Green	Waterman,Danielle C.	944-39-5991		Oak Creek	ID	1648 Maple Str
3	3	86	04/01/1928		DeSantis,Christen N.	336-13-6311		Boston	AZ	8572 Maple Str
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.

10.10 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**.

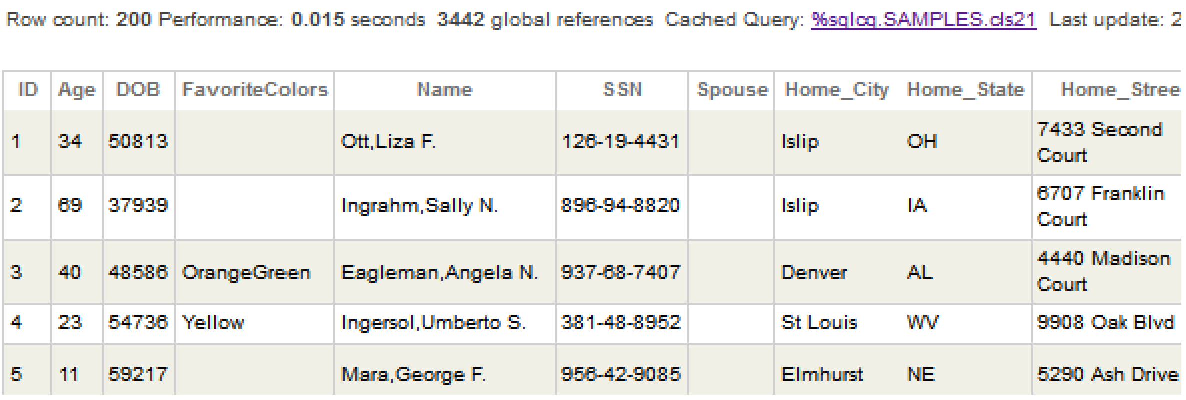
4. Type an SQL query into the input box. For example:

   ```
   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

10.11 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.

10.12 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.

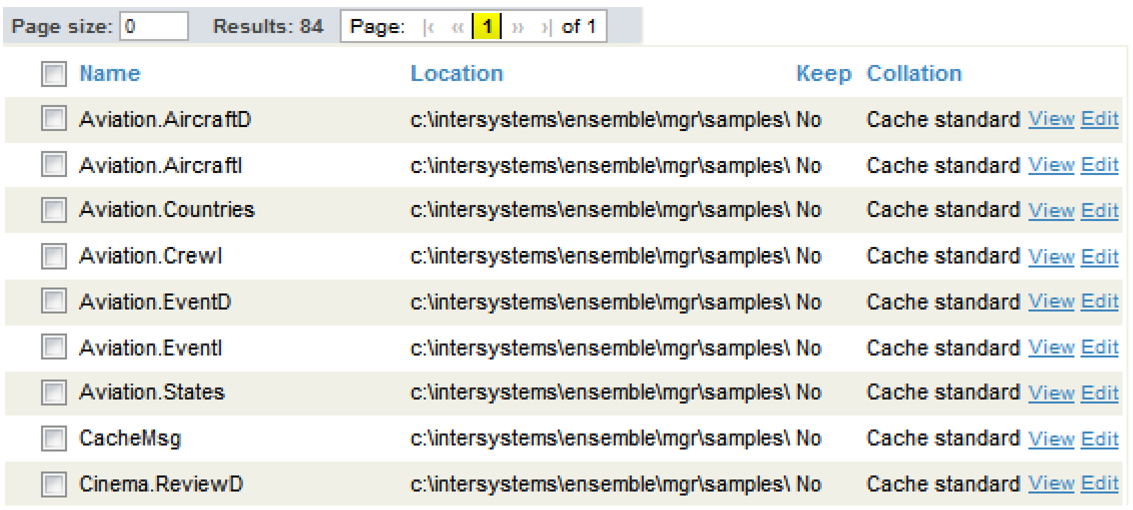
   c. Find the `<DefaultData>` element. The value between `<DefaultData>` and `</DefaultData>` is the name of the global that stores data for this class.

      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:



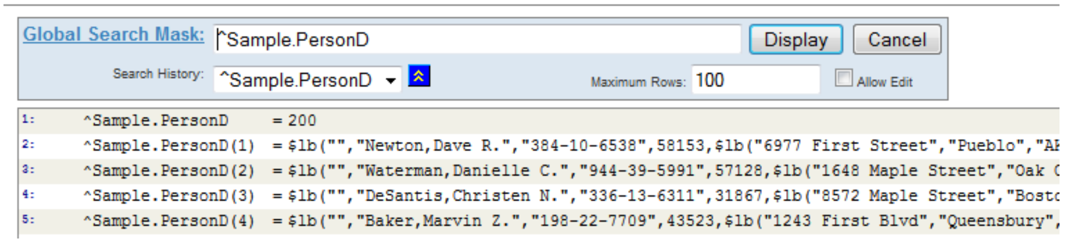
Page size: 0 Results: 84 Page: 1 of 1

	Name	Location	Keep	Collation		
☐	Aviation.AircraftD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.AircraftI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.Countries	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.CrewI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.EventD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.EventI	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Aviation.States	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	CacheMsg	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit
☐	Cinema.ReviewD	c:\intersystems\ensemble\mgr\samples\	No	Cache standard	View	Edit

   Usually most non-system globals store data for persistent classes, which means that unless you display system globals, most globals will have familiar names.

4. Select **View** in the row for the global in which you are interested.

   The system then displays the first 100 nodes of this global. For example:



Global Search Mask: ^Sample.PersonD

Search History: ^Sample.PersonD

Maximum Rows: 100 ☐ Allow Edit

```
1:     ^Sample.PersonD     = 200
2:     ^Sample.PersonD(1)  = $lb("","Newton,Dave R.","384-10-6538",58153,$lb("6977 First Street","Pueblo","A
3:     ^Sample.PersonD(2)  = $lb("","Waterman,Danielle C.","944-39-5991",57128,$lb("1648 Maple Street","Oak
4:     ^Sample.PersonD(3)  = $lb("","DeSantis,Christen N.","336-13-6311",31867,$lb("8572 Maple Street","Bost
5:     ^Sample.PersonD(4)  = $lb("","Baker,Marvin Z.","198-22-7709",43523,$lb("1243 First Blvd","Queensbury",
```

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)
```

10.13 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**.

10.14 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.)

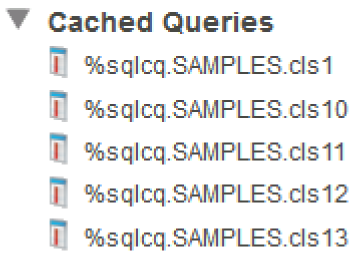
When you first execute an SQL statement, InterSystems IRIS optimizes the query and then generates and stores code that retrieves the data. It stores the code in the *query cache*, along with the optimized query text. Note that this cache is a cache of OBJ code, not of data.

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:

   Cached Queries
   %sqlcq.SAMPLES.cls1
   %sqlcq.SAMPLES.cls10
   %sqlcq.SAMPLES.cls11
   %sqlcq.SAMPLES.cls12
   %sqlcq.SAMPLES.cls13

   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**.

10.15 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**.

10.16 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).

10.17 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
Unicode Support

InterSystems IRIS supports the Unicode international character set. Unicode characters are 16-bit characters, also known as wide characters.

The $ZVERSION special variable (`Build nnnU`) and the **$SYSTEM.Version.IsUnicode()** method show that the InterSystems IRIS installation supports Unicode.

For most purposes, InterSystems IRIS only supports the Unicode Basic Multilingual Plane (hex 0000 through FFFF) which contains the most commonly-used international characters. Internally, InterSystems IRIS uses the UCS-2 encoding, which for the Basic Multilingual Plane, is the same as UTF-16. You can work with characters that are not in the Unicode Basic Multilingual Plane by using $WCHAR, $WISWIDE, and related functions.

InterSystems IRIS encodes Unicode strings into memory by allocating 16 bits (two bytes) per character, as is standard with UTF-16 encodings. However, when saving a Unicode string to a global, if all characters have numerical values of 255 or lower, InterSystems IRIS stores the string using 8 bits (one byte) per character. If the string contains characters with numerical values greater than 255, InterSystems IRIS applies a compression algorithm to reduce the amount of space the string takes up in storage.

A.1 Conversions of Data

For conversion between Unicode and UTF-8, and conversions to other character encodings, refer to the $ZCONVERT function. You can use ZZDUMP to display the hexadecimal encoding for a string of characters. You can use $CHAR to specify a character (or string of characters) by its decimal (base 10) encoding. You can use $ZHEX to convert a hexadecimal number to a decimal number, or a decimal number to a hexadecimal number.

A.2 Unicode in Identifiers

Unicode letters are alphabetic characters with decimal character code values higher than 255. For example, the Greek lowercase lambda is $CHAR(955), a Unicode letter.

Unicode letters are permitted in identifiers, with the following exceptions:

- Variable names: local variable names can contain Unicode letters. However, global variable names and process-private global names cannot contain Unicode letters. Subscripts for variables of all types can be specified with Unicode characters.
- Administrator user names and passwords used for database encryption cannot contain Unicode characters.

The locale identifier is not taken into account when dealing with Unicode characters. That is, if a identifier consisting of Unicode characters is valid in one locale, the identifier is valid in any locale. Note that the above exceptions still apply.

Note: The Japanese locale does not support accented Latin letter characters in InterSystems IRIS names. Japanese names may contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), and the Greek capital letter characters (913–929 and 931–937).

B
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.

B.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.

B.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.

B.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.

B.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.

B.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 Using Macros and Include Files.

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.

B.6 Dollar Sign ($)

This section lists forms that start with a dollar sign.

`$abcdef`

Usually, `$abcdef` is a special variable. See ObjectScript Special Variables.

`$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 Invoking Code and Passing Arguments.

`$$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 Invoking Code and Passing Arguments.

`$$$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.

B.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*.

B.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

`^||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.

^abc^def

This is an implied namespace. See ZNSPACE.

^^abcdef

This is an implied namespace. See ZNSPACE.

B.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.

{%%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.

`&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

`?abcdef`

The question mark is the pattern match operator and `abcdef` is the comparison pattern.

`@abcdef`

The at sign is the indirection operator.

B.10 See Also

- Symbols Used in ObjectScript
- Abbreviations Used in ObjectScript