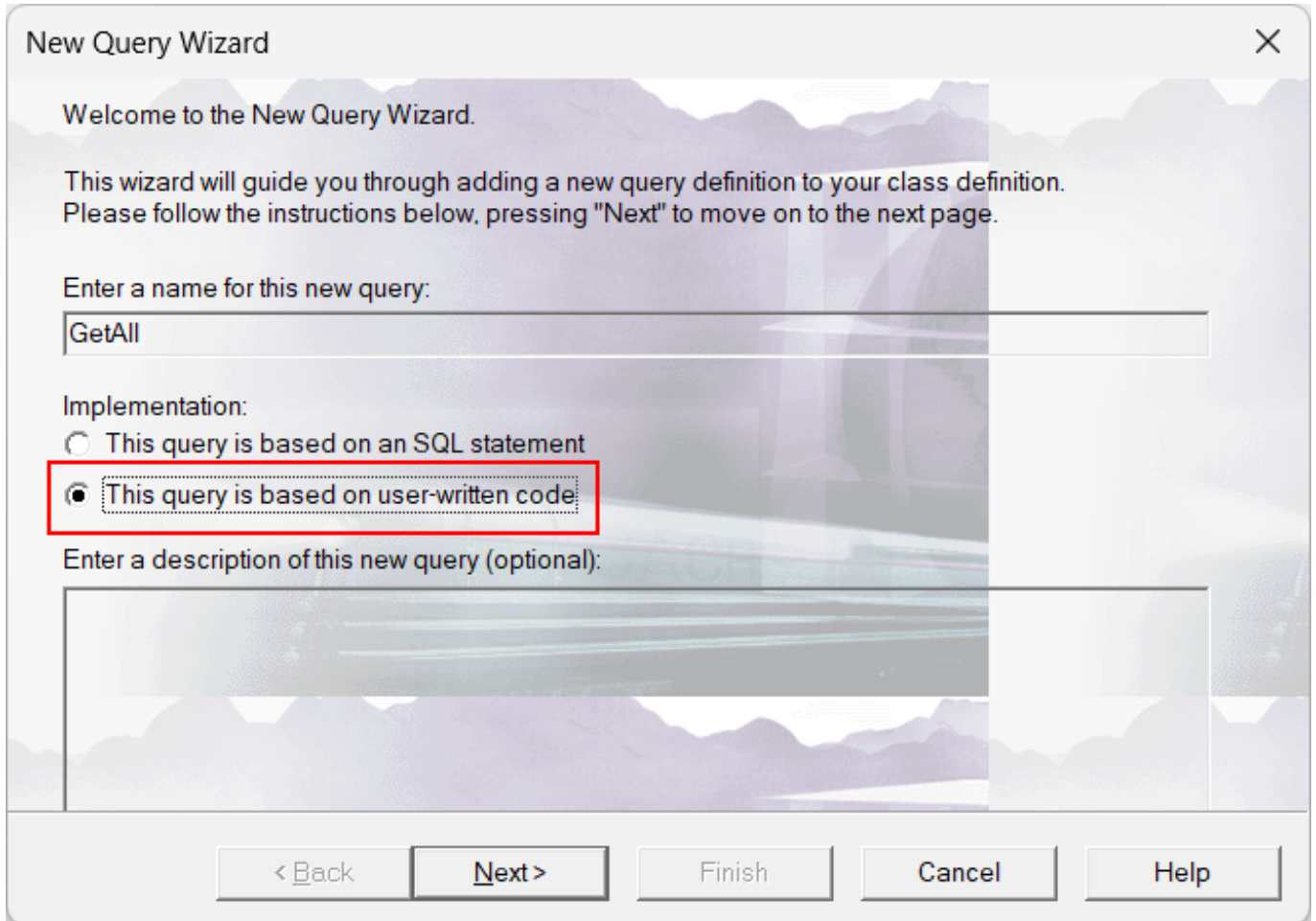


Article

[Iryna Mykhailova](#) · Mar 15, 2023 7m read

## Query as %Query or Query based on ObjectScript

In this tutorial, I ' d like to talk about Class Queries. To be more precise, about the Queries based on user-written code:



A lot of people disregard this type of query just because they aren ' t very comfortable with writing lots of ObjectScript code for the methods or they don ' t see how they can use it in their relational apps. But to be honest, for me – it ' s one of the coolest inventions for the relational model in IRIS! It lets you expose whatever information you want (not limited to tables of your database) as a relational resultset for a client. Thus, basically, you can wrap no matter what data stored in your database and beyond it into an "ordered virtual table" that your user can query. And from the point of view of a user, it will be the same resultset as if (s)he has queried a table or a view.

Not to be unfounded, here are some examples of what you can use as data for your query:

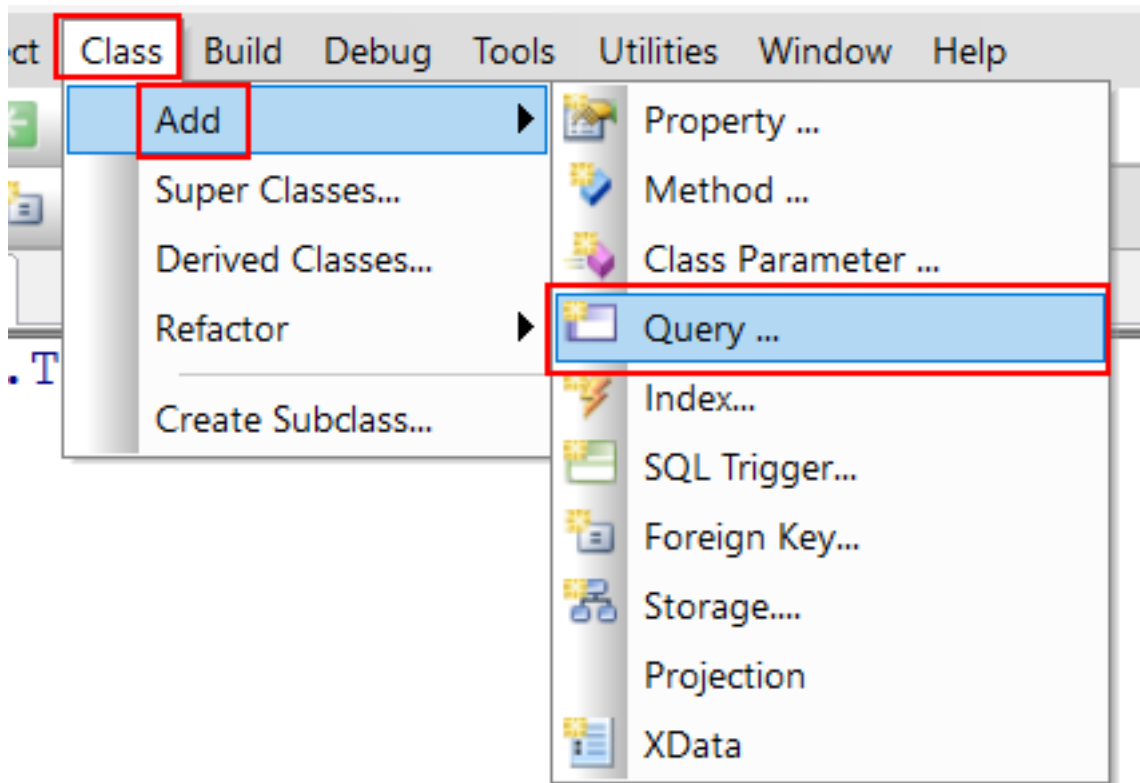
- System indicators of IRIS and your local OS (if IRIS has access to them)
- Results of external queries, like REST or SOAP (for example if you don ' t want to send requests directly from a client because of security or for any other reasons)
- Results of Embedded Cursor or Simple Statement

- Any ad hoc data that you can compose yourself
- Pretty much anything else that can come to mind and is possible to get using ObjectScript

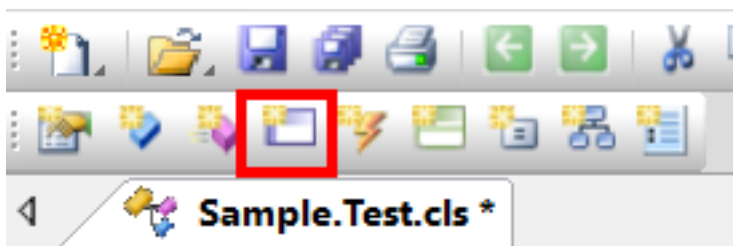
Let ' s now look at how it ' s done.

If you ' re using Studio – it ' s very nice and simple.

In your cls class you can go to the menu Class – Add – Query:



Or on toolbar “ Members ” click on the Query icon:



And the New Query Wizard will open. It will guide you through the steps of creating the necessary methods for your query to work.

Let ' s look at this process using the simplest example of wrapping the result of Embedded Cursor. Here we will actually write a query to a table but it ' s just to make it simple, further we will look at examples of using %Query to return data from other sources.

Let ' s say we have a class Sample.Human that extends %Persistent and %Populate. The latter I will use to populate data because I ' m that lazy and because I can:

```
Class Sample.Human Extends (%Persistent, %Populate)
{
```

```
Property Name As %Name;  
Property DoB As %Date(MAXVAL = 61727, MINVAL = 32507);  
Property Age As %Integer  
[ Calculated, SqlComputeCode = {set {Age} = $h - {DoB} \ 365.25}, SqlComputed ];  
}
```

When we start creating the query the first step of the Wizard is to define a name and a type:

New Query Wizard

Welcome to the New Query Wizard.

This wizard will guide you through adding a new query definition to your class definition. Please follow the instructions below, pressing "Next" to move on to the next page.

Enter a name for this new query:

GetAllOlderThan

Implementation:

☐ This query is based on an SQL statement

☒ This query is based on user-written code

Enter a description of this new query (optional):

Get all the names and ages of people whose age is greater or equal than nput parameter

< Back Next > Finish Cancel Help

The next step is to define the input parameters if there are any. To add a new parameter click on the topmost button in the righthand column of buttons:

New Query Wizard

**Input Parameters**

Please enter the name, type, and optional default value for any input parameters for this query:

#	Name	Type	Default Value
---	------	------	---------------

Argument

Name:  
Age

Type:  
%Integer

Default:  
65

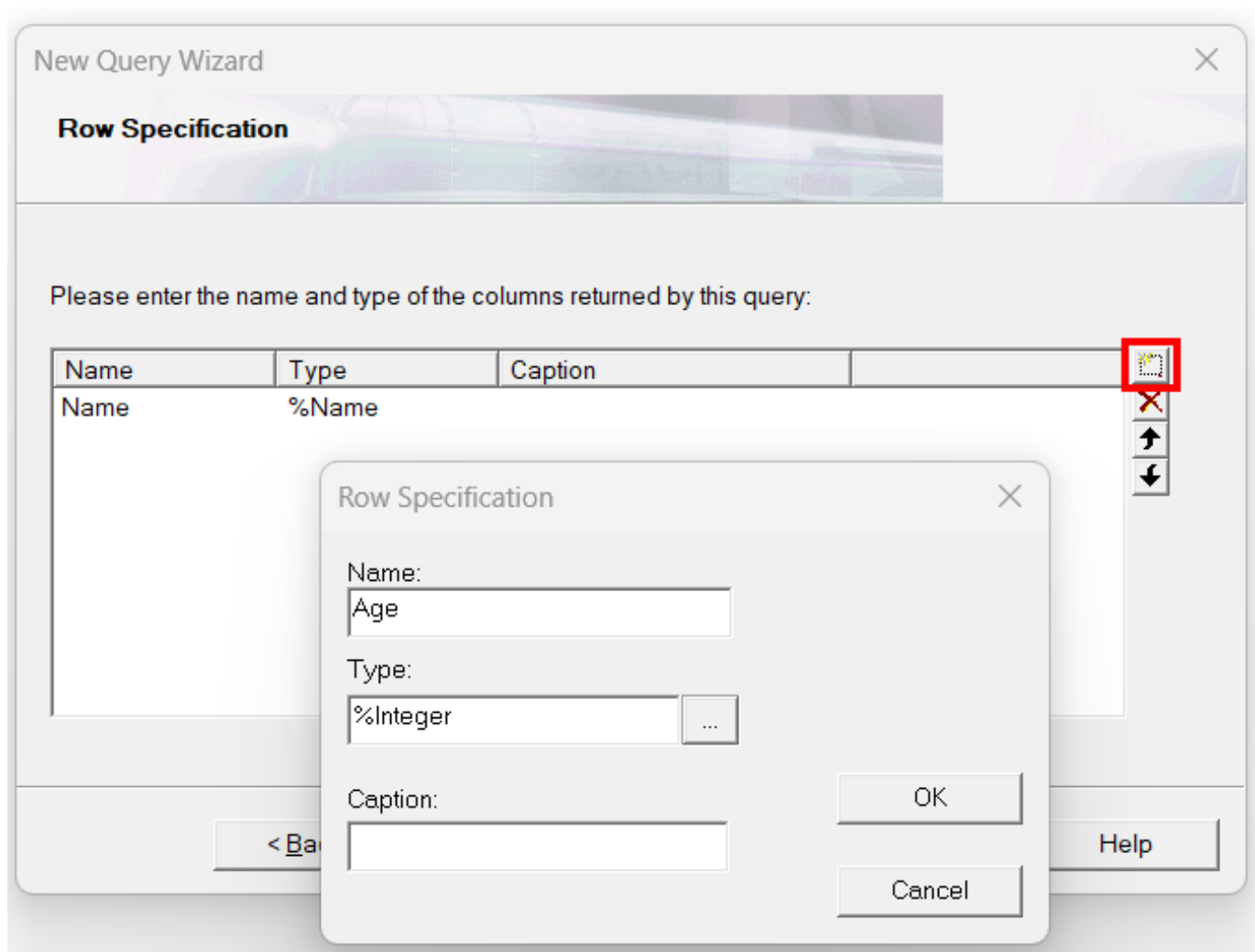
OK

Cancel

< Back

Help

The next step is a very important part – you have to define names and types or columns in a resulting resultset. When you 're creating a SQL-based query this step is done for you by the system automatically – it just looks at the columns you added in the query and takes their names and datatypes. Because in the ObjectScript based query there may be no fields whatsoever you have to tell the system yourself what to expect.



This is it. As a result of this Wizard, you will get 3 new ClassMethods and a Query in your class:

```

/// Get all the names and ages of people whose age is greater or equal than nput parameter
Query GetAllOlderThan(Age As %Integer = 65) As %Query(ROWSPEC =
"Name:%Name,Age:%Integer")
{
}

ClassMethod
  GetAllOlderThanExecute(ByRef qHandle As %Binary, Age As %Integer = 65) As %Status
{
    Quit $$$OK
}

ClassMethod GetAllOlderThanClose(ByRef qHandle As %Binary) As %Status
[ PlaceAfter = GetAllOlderThanExecute ]
{
    Quit $$$OK
}

ClassMethod GetAllOlderThanFetch(ByRef qHandle As %Binary, ByRef Row As %List
, ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = GetAllOlderThanExecute ]
{
    Quit $$$OK
}

```

}

}

The Query just tells the system that there is a query in a class that can be called by a name with a parameter and it will return 2 columns in the result. Any code you enter inside the Query will be disregarded.

All the coding is done in the ClassMethods. They are responsible for populating our “ virtual ” resultset (<QueryName>Execute – called when an SQL statement is executed), returning the next row (<QueryName>Fetch) and cleaning up after itself (<QueryName>Close). They have one parameter in common – **ByRef qHandle As %Binary** in which the data is stored and transmitted between methods. As you can see it ’ s binary data so you can put anything (common sense is your limit) inside. In <QueryName>Execute there is also an input parameter of the Query: in this case, it ’ **Age As %Integer = 65**. And in <QueryName>Fetch there are two additional parameters:

- **ByRef Row As %List** – it ’ s a %List that contains the current row of the “ virtual ” resultset with the number and values of the columns described in the Query ’ s ROWSPEC (**Name:%Name, Age:%Integer**).
- **ByRef AtEnd As %Integer = 0** – this is a flag that signals that the current row is the last row (1) or not (0).

If you ’ re working in VSCode, you ’ ll have to write all the signatures yourself and try not to make any mistakes

Now that we know which parameter is responsible for what, let ’ s look at the code:

```
/// Get all the names and ages of people whose age is greater or equal than nput parameter
```

```
Query GetAllOlderThan(Age As %Integer = 65) As %Query(ROWSPEC =
"Name:%Name,Age:%Integer") [ SqlProc ]
{
}
```

```
ClassMethod
```

```
GetAllOlderThanExecute(ByRef qHandle As %Binary, Age As %Integer = 65) As %Status
{
    &sql(DECLARE C CURSOR FOR
        SELECT Name, Age
        FROM Sample.Human
        WHERE Age >= :Age
    )
    &sql(OPEN C)
    Quit $$$OK
}
```

```
ClassMethod GetAllOlderThanClose(ByRef qHandle As %Binary) As %Status
[ PlaceAfter = GetAllOlderThanExecute ]
{
    &sql(CLOSE C)
    Quit $$$OK
}
```

```
ClassMethod GetAllOlderThanFetch(ByRef qHandle As %Binary, ByRef Row As %List
, ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = GetAllOlderThanExecute ]
{
    &sql(FETCH C INTO :Name, :Age)
```

```

    If (SQLCODE'=0) {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    Set Row = $Lb(Name, Age)
    Quit $$$OK
}

```

This code declares and opens an Embedded Cursor that selects the names and ages of people older than **Age** in <QueryName>Execute, then fetches the result from the cursor and forms a list in <QueryName>Fetch and closes it in <QueryName>Close.

We can call it using code:

```

SET tStatement = ##class(%SQL.Statement).%New()
SET rstatus = tStatement.%PrepareClassQuery("Sample.Human", "GetAllOlderThan")
SET rset = tStatement.%Execute()
DO rset.%Display()

```

Very nice and neat. And probably not what you were looking for.

As an example of usage without data stored in the database, let ' s say that for some reason we don ' t have access to the actual tables, but we need to check that our application works as it should. We need the query to return some test data. We can rewrite this example so that the data is generated automatically during the fetching of a new row.

Obviously, we don ' t need to save data in the memory so there is no need to populate the **qHandle** variable in <QueryName>Execute – we can do the creation of data inside <QueryName>Fetch. And in **qHandle** we will store the size of the resultset to return (a random number less than 200, for example) and the number of the current row which we will increment in <QueryName>Fetch. In the end, we will get the code:

```

Query GetAllOlderThan(Age As %Integer = 65) As %Query(ROWSPEC =
"Name:%Name,Age:%Integer") [ SqlProc ]
{
}

ClassMethod
    GetAllOlderThanExecute(ByRef qHandle As %Binary, Age As %Integer = 65) As %Status
{
    set qHandle = $lb($random(200), 0, Age)
    Quit $$$OK
}

ClassMethod GetAllOlderThanClose(ByRef qHandle As %Binary) As %Status
[ PlaceAfter = GetAllOlderThanExecute ]
{
    set qHandle = ""
    Quit $$$OK
}

ClassMethod GetAllOlderThanFetch(ByRef qHandle As %Binary, ByRef Row As %List
, ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = GetAllOlderThanExecute ]
{
    if $ListGet(qHandle, 2) = $ListGet(qHandle, 1)
    {

```



```
Set Row = ""
set AtEnd = 1
} else
{
    Set Row = $Lb(##class(%PopulateUtils).Name(), ##class(
%PopulateUtils).Integer($ListGet(qHandle, 3), 90))
    set $list(qHandle, 2) = $list(qHandle, 2) + 1
}
Quit $$$OK
}
}
```

As you can see, I ' m generating data ad hoc. This means that you can get your data from wherever, wrap it into a resultset and make it accessible from your app that uses ODBC/JDBC outside the IRIS database. Which in return means that you can use the usual relational access to get non-relational data if you manage to structure it.

[#Best Practices](#) [#SQL](#) [#Tutorial](#) [#InterSystems IRIS](#)

---

Source URL: <https://community.intersystems.com/post/query-query-or-query-based-objectscript>