Article
Eduard Lebedyuk · Jul 16, 2019 4m read

Classes, tables and globals - how it all works?

When I describe InterSystems IRIS to more technically-minded people, I always start with how it is a multimodel DBMS at its core.

In my opinion that is its main advantage (on the DBMS side). And the data is stored only once. You just choose the access API you want to use.

- You want some sort of summary for your data? Use SQL!
- Do you want to work extensively with one record? Use objects!
- Want to access or set one value and you know the key? Use globals!

On first blush it's a nice story - short and concise and it gets the message across, but when people really start working with InterSystems IRIS the questions start. How are classes and tables and globals related? What are they to each other? How's the data really stored?

In this article I would try to answer these questions and explain what's really going on.

Part one. Model bias.

People working with data are often biased towards the model they work with.

Developers think in objects. For them databases and tables are boxes you interact with via CRUD (Create-Read-Update-Delete, preferably over ORM) but the underlying conceptual model is objects (of course it's mainly true for developers in object-oriented languages - so most of us).

On the other hand, as a consequence of spending so much time in relational DBMSes, DBAs often think of data as tables. Objects are just wrappers over rows in this case.

And with InterSystems IRIS, a persistent class is also a table, which stores data in global, so some clarification is required.

Part two. An example.

Let's say you created class Point:

```
Class try.Point Extends %Persistent [DDLAllowed]
{
    Property X;
    Property Y;
}
```

You can also create the same class with DDL/SQL:

```
CREATE Table try.Point (
```

```
    X VARCHAR(50),
    Y VARCHAR(50))
```

After compilation, our new class would have auto-generated a storage structure which maps data that is natively stored in globals to columns (or properties if you're an object-oriented thinker):

```
Storage Default
{
<Data name="PointDefaultData">
    <Value name="1">
        <Value>%%CLASSNAME</Value>
    </Value>
    <Value name="2">
        <Value>X</Value>
    </Value>
    <Value name="3">
        <Value>Y</Value>
    </Value>
</Data>
<DataLocation>^try.PointD</DataLocation>
<DefaultData>PointDefaultData</DefaultData>
<IdLocation>^try.PointD</IdLocation>
<IndexLocation>^try.PointI</IndexLocation>
<StreamLocation>^try.PointS</StreamLocation>
<Type>%Library.CacheStorage</Type>
}
```

What is going on here?

From the bottom-up (**bold** words are important, ignore the rest):

- Type: generated storage type, in our case the default storage for persistent objects
- StreamLocation - global where we store streams
- IndexLocation - global for indices
- IdLocation - global where we store ID autoincremental counter
- **DefaultData** - storage XML element which maps global value to columns/properties
- **DataLocation** - global in which to store data

Now our "DefaultData" is PointDefaultData so let's look closer at its structure. Essentially it says that global node has this structure:

- 1 - %%CLASSNAME
- 2 - X
- 3 - Y

So we might expect our global to look like this:

```
^try.PointD(id) = %%CLASSNAME, X, Y
```

But if we output our global it would be empty because we didn't add any data:

```
zw ^try.PointD
```

Let's add one object:

```
set p = ##class(try.Point).%New()
set p.X = 1
set p.Y = 2
write p.%Save()
```

And here's our global

```
zw ^try.PointD
^try.PointD=1
^try.PointD(1)=$lb("",1,2)
```

As you see our expected structure %%CLASSNAME, X, Y is set with $lb("",1,2) which corresponds to X and Y properties of our object (%%CLASSNAME is system property, ignore it).

We can also add a row via SQL:

```
INSERT INTO try.Point (X, Y) VALUES (3,4)
```

Now our global looks like this:

```
zw ^try.PointD
^try.PointD=2
^try.PointD(1)=$lb("",1,2)
^try.PointD(2)=$lb("",3,4)
```

So the data we add via objects or SQL are stored in globals according to storage definitions (side note: you can manually modify the storage definition by replacing X and Y in PointDefaultData - check what happens to the new data!).

Now, what happens when we want to execute a SQL query?

```
SELECT * FROM try.Point
```

It is translated into ObjectScript code that iterates over the ^try.PointD global and populates columns based on the storage definition - the PointDefaultData part of it precisely.

Now for modifications. Let's delete all the data from the table:

```
DELETE FROM try.Point
```

And let's see our global at this point:

```
zw ^try.PointD
^try.PointD=2
```

Note that only ID counter is left, so new object/row would have an ID=3. Also our class and table continue to exist.

But what happens when we run:

```
DROP TABLE try.Point
```

It would destroy the table, the class, and delete the global.

```
zw ^try.PointD
```

If you followed this example, I hope you now have a better understanding of how globals, classes and tables integrate and complement each other. Using the right API for the job at hand results in faster, more agile, less buggy development.

#Beginner #Best Practices #Globals #Object Data Model #Relational Tables #SQL #InterSystems IRIS