Article
Joel Solon · Sep 6, 2016 3m read

These Are NOT Identical Twins: the Class Keywords DependsOn and CompileAfter

In most cases, if one class depends on another, the class compiler will detect this and determine the correct compilation order. For example:

- Compiling a superclass triggers compilation of its subclasses.
- Compiling a custom datatype class triggers compilation of any classes with a property of that type.

Yet sometimes, a developer needs to specify compilation order. InterSystems class definitions contain two keywords for this, DependsOn and CompileAfter, that are very similar but **not** identical. They are useful during development, and also when importing and compiling a set of classes for the first time.

The documentation for these keywords is here:

- http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=...
- http://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=...

Here's how the keywords look in a class definition:

```
Class Test.B [ DependsOn = Test.A ]
Class Test.D [ CompileAfter = Test.C ]
```

When you compile Test.B, if Test.A has been changed and saved but not compiled, the compiler automatically adds Test.A to the list of classes to compile, and compiles class Test.A before class Test.B. The same thing happens when compiling Test.D: Test.C is compiled before Test.D.

So what's the difference between them? To understand the difference, you must first understand that the compiler is responsible for compilation (checking class for errors, resolving inheritance, and so on) **and** generating code.

- Using DependsOn, the compiler compiles and generates code for Test.A before compiling and generating code for Test.B. This is necessary when Test.B calls code in Test.A at compile time. The documentation refers to this as making Test.A *runnable* before compiling Test.B.
- Using CompileAfter, the compiler compiles Test.C before Test.D, but it can generate the code for these classes in any order.

As you can see, DependsOn encompasses CompileAfter.

Let's look at some examples. If you have your own examples to share, please do so in the comments.

Example #1. Test.B **depends on** Test.A. The value of the STRING parameter is computed at **compile time** by calling Test.A and accessing its TEXT parameter. When Test.B is compiled (even if compiled alone), the compiler compiles Test.A and generates the code for it, and then compiles and generates code for Test.B. Note that changing Test.A and compiling it **does not** trigger compilation of Test.B.

```
Class Test.A
{
Parameter TEXT = "this is some text";
}

Class Test.B [ DependsOn = Test.A ]
{
Parameter STRING = {##class(Test.A).#TEXT};

ClassMethod M1()
{
    write ..#STRING, !
}
}
```

Example #2. Test.D should be **compiled after** Test.C. If Test.D is compiled (even if compiled alone), the compiler will compile Test.C and then Test.D. Later, when a Test.D object is created, the constructor can safely call the M2() method to initialize the value for the Test property. As before, note that changing Test.C and compiling it **does not** trigger compilation of Test.D.

```
Class Test.C
{
ClassMethod M2() as %Integer
{
    return 4
}
}

Class Test.D extends %Persistent [ CompileAfter = Test.C ]
{
Property Test As %Integer [ InitialExpression = {##class(Test.C).M2()} ];
}
```

Example #3. Test.F should be **compiled after** Test.E. Test.F contains embedded SQL that references columns from the Test.E table. If Test.F is compiled (even if compiled alone), the compiler will compile Test.E and then Test.F. This example is a little different than the first two because embedded SQL dependency is **stored**. So you could remove the CompileAfter keyword after the first time you compile these two classes, and they would continue to compile in the correct order. But including it guarantees that if these two classes are imported and compiled on another system, they will always compile in the correct order.

```
Class Test.E extends %Persistent
{
Property Name as %String;

Property Phone as %String;
}

Class Test.F [ CompileAfter = Test.E ]
{
ClassMethod M3()
{
    &sql(select Name, Phone into :nm, :ph from Test.E)
    write !, nm, "  ", ph
}
}
```

#Compiler #Object Data Model #Caché