Class Definition Reference

Version 2019.1
2019-05-23

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

Class Definition Reference
InterSystems IRIS Data Platform Version 2019.1 2019-05-23

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book provides reference information on the structure of and elements used in class definitions. It discusses the following:

- Class Definitions
- Class Keywords
- ForeignKey Keywords
- Index Keywords
- Method Keywords
- Parameter Keywords
- Projection Keywords
- Property Keywords
- Query Keywords
- Trigger Keywords
- XData Keywords
- Storage Keywords

For a detailed outline, see the table of contents.

The following books provide related information:

- *Orientation Guide for Server-Side Programming*
- *Defining and Using Classes*
- *Using InterSystems SQL*

Class Definitions

This reference formally describes the structure of class definitions.

For information on defining classes, see “Defining and Compiling Classes” in *Defining and Using Classes*.

Class Definitions

Describes the structure of a class definition.

Introduction

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.

Details

A class definition has the following structure:

```
Import import_ppackage_list
Include include_code
IncludeGenerator include_generator_code

/// description
Class package.shortclassname Extends superclass_list [ class_keyword_list ]
{
Class_members
}
```

Where:

- *import_package_list* (optional) specifies the names of any packages that you wish your class to import from. This affects how the compiler resolves short class names; see "Importing Packages" in *Defining and Using Classes*.

  This option, if specified, is either the name of a single package or is a comma-separated list of multiple packages, enclosed in parentheses.

  If *import_package_list* is null, do not add the `Import` line at the start of the class definition.

  Note: If a class imports any packages, that class does *not* automatically import the User package.

  Imported packages are inherited from all superclasses. If a subclass specifies one or more import packages, those are added to any import packages defined by the superclasses.

- *include_code* (optional) specifies InterSystems IRIS include (.inc) files used when compiling this class.

  This option, if specified, is either the name of a single include file (without the .inc extension) or is a comma-separated list of multiple include files, enclosed in parentheses.

  If *include_code* is null, omit the `Include` line at the start of the class definition.

  For an introduction to include files, see "Include Files" in the *InterSystems Programming Tools Index*.

  Include files are inherited from all superclasses. If a subclass specifies one or more include files, those are added to any include files defined by the superclasses.

- *include_generator_code* (optional) specifies InterSystems IRIS include (.inc) files used when compiling the generator methods for this class. For information on generator methods, see "Defining Method and Trigger Generators" in *Defining and Using Classes*.

  For general comments, see the previous item.

  If *include_generator_code* is null, do not add the `IncludeGenerator` line at the start of the class definition.

- *description* (optional) is intended for display in the Class Reference. A description may consist of multiple lines and may include HTML formatting tags and additional tags such as `<class>` and `<method>`. For limitations and details, see "Creating Class Documentation" in *Defining and Using Classes*. The description is blank by default.

- *package* (required) is a valid package name, as described in "Package Names" in *Defining and Using Classes*.

- *shortclassname* (required) is a valid class name. Together, *package* and *shortclassname* form the complete class name, which is subject to a length restriction. See "Naming Conventions" in *Defining and Using Classes*.
- *superclass_list* (optional) specifies the class or classes from which this class inherits. This option, if specified, is either the name of a single class (without the .cls extension) or is a comma-separated list of classes, enclosed in parentheses.

  The first such class is referred to as the *primary superclass*; any additional classes are secondary superclasses. For information, see "Inheritance" in *Defining and Using Classes*.

  If *superclass_list* is null, omit the word `Extends` from the class definition.
- *class_keyword_list* (optional) is a comma-separated list of keywords that (in most cases) affect how the compiler generates code for this class definition.

  See the section "Class Keywords."

  If this list is omitted, also omit the square brackets.
- *Class_members* is zero or more definitions of class members.

  For an introduction, see "Kinds of Class Members" in *Defining and Using Classes*. The other sections in this reference provide details on these kinds of class member.

See Also

- Defining and Compiling Classes in *Defining and Using Classes*
- Class Keywords
- Foreign Key Definitions
- Index Definitions
- Method Definitions
- Parameter Definitions
- Projection Definitions
- Property Definitions
- Query Definitions
- Trigger Definitions
- XData Blocks

This reference does not formally introduce storage definitions. For an introduction to storage definitions, see "Storage" in "Persistent Objects and InterSystems IRIS SQL" in the *Orientation Guide for Server-Side Programming*. Also see "Storage Definitions and Storage Classes" in "Defining Persistent Classes" in *Defining and Using Classes*.

Foreign Key Definitions

Describes the structure of a foreign key definition.

Introduction

A foreign key defines a referential integrity constraint. When a table containing a foreign key constraint is modified, the foreign key constraints are checked.

You can add foreign key definitions to persistent classes. They are not meaningful in other kinds of classes.

Note that you can also enforce reference integrity by defining relationship properties that connect the classes.

Details

A foreign key definition has the following structure:

```
/// description
ForeignKey name(key_props) References referenced_class(ref_index) [ keyword_list ];
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the foreign key. This must be a valid class member name, and must not conflict with any other class member names.
- *key_props* (required) specifies the property or properties that are constrained by this foreign key. Specifically this property or properties must match the referenced value in the foreign table.

  This is a comma-separated list of property names.

  These properties must be in the same class that defines the foreign key.
- *referenced_class* (required) specifies the foreign table (that is, the class to which the foreign key points).
- *ref_index* (optional) specifies the unique index name within *referenced_class*.

  If you omit *ref_props*, then the system uses the IDKEY index in *referenced_class*.
- *keyword_list* (optional) is a comma-separated list of keywords that further define the foreign key.

  See the section "Foreign Key Keywords."

  If this list is omitted, also omit the square brackets.

Example

```
ForeignKey EmpKey(EmpId) References MyApp.Employee(EmpID) [ OnDelete = cascade ];
```

See Also

- "Using Foreign Keys" in *Using InterSystems SQL*
- "Foreign Key Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Index Definitions

Describes the structure of an index definition.

Introduction

An index is a structure maintained by a persistent class that is intended to be used to optimize queries and other functions. These indices are automatically maintained whenever INSERT, UPDATE, or DELETE SQL-based operations are carried out against the database; likewise for object-based actions. The SQL Query Processor makes use of available indices when preparing and executing SQL queries.

You can add index definitions to persistent classes. They are not meaningful in other kinds of classes.

Details

An index definition has the following structure:

```
/// description
Index name On property_expression_list [ keyword_list ];
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the index. This must be a valid identifier that is unique within this class or table.

  This name is used for database administrative purposes (reporting, index building, dropping indices, and so on).
- *property_expression_list* (required) specifies the property or properties on which the index is based and may also include a collation specification for each property. This option is either a single property expression or a comma-separated list of property expressions, enclosed in parentheses.

  A given property expression consists of:

  – The name of the property to be indexed.

  – An optional `(ELEMENTS)` or `(KEYS)` expression, which provides a means of indexing on collection subvalues.

  – An optional collation expression.

  For more details, see "Defining and Building Indices" in the *SQL Optimization Guide*.
- *keyword_list* (optional) is a comma-separated list of keywords that further define the index.

  See the section "Index Keywords."

  If this list is omitted, also omit the square brackets.

For example, the following class definition defines two properties and an index based on each of them:

```
Class MyApp.Student Extends %Persistent
{

Property Name As %String;

Property GPA As %Double;

Index NameIDX On Name;

Index GPAIDX On GPA;
}
```

See Also

- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Index Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Method Definitions

Describes the structure of a method definition.

Introduction

In most cases, a method definition defines the runtime behavior of the method. InterSystems IRIS supports also method generators, which are a special kind of method that generate the code that is used at runtime.

Details

A method definition has the following structure:

```
/// description
Method name(formal_spec) As returnclass [ keyword_list ]
{ implementation }
```

Or (for a class method):

```
/// description
ClassMethod name(formal_spec) As returnclass [ keyword_list ]
{ implementation }
```

Or (for a client method):

```
/// description
ClientMethod name(formal_spec) As returnclass [ keyword_list ]
{ implementation }
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the method. This must be a valid class member name, and must not conflict with any other class member names.
- *formal_spec* (optional) specifies the list of arguments that are passed to or from the method.

  The formal specification is a list of a method's arguments, their types, their call-type (ByRef, Output, or ByVal), and optional default values. The Output call type is used to indicate arguments which are passed by reference, but whose incoming value is nominally not used.
- *returnclass* (optional) specifies the type of value returned by this method, if any. If you omit *returnclass*, also omit the word As
- *keyword_list* (optional) is a comma-separated list of keywords that further define the method.

  See the section "Method Keywords."

  If this list is omitted, also omit the square brackets.
- *implementation* (optional) is zero or more lines of code that define what the method does.

  To specify the programming language used, use the class-level Language or method-level Language keyword.

Parameter Values

For *formal_spec* and *returnclass*, you can specify optional parameter values after the class names. If the method is used as an SQL stored procedure, then these parameter values are used to provide additional information to an ODBC or JDBC client. These parameters are ignored in all other cases. For example:

```
ClassMethod MyProc(data As %String(MAXLEN = 85)) As %Integer [ SQLProc ]
{
  Quit 22
}
```

For another example:

```
ClassMethod GetName() As %String(MAXLEN=222) [ SQLProc ]
{
  Quit "Mo"
}
```

See Also

- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Method Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Parameter Definitions

Describes the structure of a parameter definition.

Introduction

A parameter definition defines a constant value available to all objects of a given class. When you create a class definition (or at any point before compilation), you can set the values for its class parameters. By default, the value of each parameter is the null string, but you can specify a non-null value as part of the parameter definition. At compile-time, the value of the parameter is established for all instances of a class. With rare exceptions, this value cannot be altered at runtime.

Details

A parameter definition has the following structure:

```
/// description
Parameter name As parameter_type [ keyword_list ] = value ;
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the parameter. This must be a valid class member name, and must not conflict with any other class member names.
- *parameter_type* (optional) specifies the user interface type of the parameter.

  This is not a class name; see the next section. This information is intended for use by development tools, to assist the developer who provides a value for the parameter in a subclass. In most cases, the compiler ignores this keyword.

  If you omit *parameter_type*, also omit the word `As`
- *value* (optional) specifies the value of the parameter. If you omit *value*, also omit the equals sign `=`
- *keyword_list* (optional) is a comma-separated list of keywords that further define the parameter.

  See the section "Parameter Keywords."

  If this list is omitted, also omit the square brackets.

Allowed Types for Parameters

The *parameter_type* option can be one of the following values:

- `BOOLEAN` — A true (1) or false (0) value.
- `CLASSNAME` — A valid class name.
- `COSCODE` — ObjectScript code.
- `COSEXPRESSION` — A valid ObjectScript expression.

  If a parameter is of type `COSEXPRESSION`, the expression is evaluated at runtime.

  Unlike most other values of the parameter Type keyword, this value affects the compiler.
- `COSIDENTIFIER` — A valid ObjectScript identifier.
- `INTEGER` — An integer value.
- `SQL` — An SQL statement.
- `SQLIDENTIFIER` — A valid SQL identifier.

- `STRING` — A string value.
- `TEXT` — A multi-line text value.
- `CONFIGVALUE` — A parameter that can be modified outside of the class definition. Unlike most other values of the parameter Type keyword, this value affects the compiler. If a parameter is of type `CONFIGVALUE`, then you can modify the parameter via the **$SYSTEM.OBJ.UpdateConfigParam()**. For example, the following changes the value of the parameter *MYPARM* (in the class MyApp.MyClass so that its new value is 42:

  ```
  set sc=$system.OBJ.UpdateConfigParam("MyApp.MyClass","MYPARM",42)
  ```

  Note that **$SYSTEM.OBJ.UpdateConfigParam()** affects the generated class descriptor as used by any new processes, but does not affect the class definition. If you recompile the class, InterSystems IRIS regenerates the class descriptor, which will now use the value of this parameter as contained in the class definition (thus overwriting the change made via **$SYSTEM.OBJ.UpdateConfigParam()**).

You also can omit *parameter_type*, in which case Studio will allow any value for the parameter.

Example

```
/// This is the name of our web service.
Parameter SERVICENAME = "SOAPDemo" ;
```

See Also

- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Parameter Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Projection Definitions

Describes the structure of a projection definition.

Introduction

A projection definition instructs the class compiler to perform specified operations when a class definition is compiled or removed. A projection defines the name of a projection class (derived from the %Projection.AbstractProjection class) that implements methods that are called when the compilation of a class is complete and when a class definition is removed (either because it is being deleted or because the class is about to be recompiled).

Details

A projection definition has the following structure:

```
/// description
Projection name As projection_class (parameter_list) ;
```

Where:

- *description* (optional) is intended for display in the Class Reference (but note that projections are not currently shown in the Class Reference). The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the projection. This must be a valid class member name, and must not conflict with any other class member names.
- *projection_class* (required) is the name of the projection class, which is a subclass of %Projection.AbstractProjection.
- *parameter_list* (optional) is a comma-separated list of parameters and their values. If specified, these should be parameters used by *projection_class*.

  If this list is omitted, also omit the parentheses.
- *keyword_list* (optional) is a comma-separated list of keywords that further define the projection.

  See the section "Projection Keywords."

  If this list is omitted, also omit the square brackets.

See Also

- "Projection Keywords" in this book
- "Defining Class Projections" in *Defining and Using Classes*
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Property Definitions

Describes the structure of a property definition. Note that a relationship is a property.

Introduction

A property contains information relevant to an instance of a class. You can add property definitions to object classes. They are not meaningful in other kinds of classes.

Details

A property definition has the following structure:

```
/// description
Property name As classname (parameter_list) [ keyword_list ] ;
```

Or (for a list property):

```
/// description
Property name As List Of classname (parameter_list) [ keyword_list ] ;
```

Or (for an array property):

```
/// description
Property name As Array Of classname (parameter_list) [ keyword_list ] ;
```

Or (for a relationship property):

```
/// description
Relationship name As classname [ keyword_list ] ;
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the property. This must be a valid class member name, and must not conflict with any other class member names.
- *classname* (optional) is the name of the class on which this property is based.
- *parameter_list* (optional) is a comma-separated list of parameters and their values. If specified, these should be either parameters used by *classname* or parameters that are available to all properties.

  If this list is omitted, also omit the parentheses.
- *keyword_list* (required for a relationship property but otherwise optional) is a comma-separated list of keywords that further define the property.

  See the section "Property Keywords."

  If this list is omitted, also omit the square brackets.

Example

```
/// Person's Social Security number.
Property SSN As %String(PATTERN = "3N1""-""2N1""-""4N") [ Required  ];
```

See Also

- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*

- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Property Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Query Definitions

Describes the structure of a query definition.

Introduction

A class query is a named query that is part of a class structure and that can be accessed via dynamic SQL.

You can define class queries within any class; there is no requirement to contain them within persistent classes.

Details

A query definition has the following structure:

```
/// description
Query name(formal_spec) As classname [ keyword_list ]
{ implementation }
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the query. This must be a valid class member name, and must not conflict with any other class member names.
- *formal_spec* (optional) specifies the list of arguments that are passed to the query.

  Specifically, this is the list of arguments that are passed to the query via the **Execute()** method of the associated query class.

  See the comments for *formal_spec* in "Method Definitions."
- *classname* (required) specifies the query class used by this query.

  This is typically %SQLQuery for SQL-based queries and %Query for custom queries. See "Class Queries" in *Defining and Using Classes*.
- *keyword_list* (optional) is a comma-separated list of keywords that further define the query.

  See the section "Query Keywords."

  If this list is omitted, also omit the square brackets.
- *implementation* (optional) is zero or more lines of code that define the query.

See Also

- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Method Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Trigger Definitions

Describes the structure of a trigger definition.

Introduction

Triggers are code segments executed when specific events occur in InterSystems SQL. InterSystems IRIS supports triggers based on the execution of INSERT, UPDATE, and DELETE commands. The specified code will be executed either immediately before or immediately after the relevant command is executed, depending on the trigger definition. Each event can have multiple triggers as long as they are assigned an execution order.

You can add trigger definitions to persistent classes. They are not meaningful in other kinds of classes.

Details

A trigger definition has the following structure:

```
/// description
Trigger name [ keyword_list ]
{ implementation }
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the trigger. This must be a valid class member name, and must not conflict with any other class member names.
- *keyword_list* (required) is a comma-separated list of keywords that further define the trigger.

  See the section "Trigger Keywords."
- *implementation* (required) is zero or more lines of ObjectScript code that define the code that is to be executed when the trigger is fired.

Example

```
/// This trigger updates the LogTable after every insert
Trigger LogEvent [ Event = INSERT, Time = AFTER ]
{
    // get row id of inserted row
    NEW id
    SET id = {ID}

    // INSERT value into Log table
    &sql(INSERT INTO LogTable (TableName, IDValue) VALUES ('MyApp.Person', :id))

}
```

See Also

- "Using Triggers" in *Using InterSystems SQL*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Trigger Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

XData Blocks

Describes the structure of an XData block.

Introduction

An XData block is a named unit of data that you include in a class definition, typically for use by a method in the class. Most frequently, it is a well-formed XML document, but it could consist of other forms of data, such as JSON or YAML.

Details

An XData block has the following structure:

```
/// description
XData name [ keyword_list ]
{
data
}
```

Where:

- *description* (optional) is intended for display in the Class Reference. The description is blank by default. See "Creating Class Documentation" in *Defining and Using Classes*.
- *name* (required) is the name of the XData block. This must be a valid class member name, and must not conflict with any other class member names.
- *data* (optional) contains the payload of the XData block. If XML, it must be a well-formed document (with a single root element), without the XML declaration at its start.
- *keyword_list* (optional) is a comma-separated list of keywords that further define the XData block.

  See the section "XData Keywords."

  If this list is omitted, also omit the square brackets.

Example

```
Class Demo.CoffeeMakerRESTServer Extends %CSP.REST
 {
  Parameter HandleCorsRequest = 1

  XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
  {
    <Routes>
      <Route Url="/test" Method="GET" Call="test"/>
      <Route Url="/coffeemakers" Method="GET" Call="GetAll" />
      <Route Url="/coffeemaker/:id" Method="GET" Call="GetCoffeeMakerInfo" />
      <Route Url="/newcoffeemaker" Method="POST" Call="NewMaker" />
      <Route Url="/coffeemaker/:id" Method="PUT" Call="EditMaker" />
      <Route Url="/coffeemaker/:id" Method="DELETE" Call="RemoveCoffeemaker"/>
   </Routes>
  }
```

See Also

- "Defining and Using XData Blocks" in *Defining and Using Classes*
- "XData Keywords" in this book
- "Class Limits" in "General System Limits" in the *Orientation Guide for Server-Side Programming*

Class Keywords

This reference describes the keywords that apply to a class as a whole or that specify the default behavior of its members. Later reference sections describe the keywords that apply to specific class members.

For general information on class definitions, see “Class Definitions.”

Abstract

Specifies whether this is an *abstract* class.

Usage

To mark a class as abstract, use the following syntax:

```
Class MyApp.MyClass [ Abstract ]
{ //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If a class is *abstract*, you cannot create instances of it.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the class is not abstract.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ClassType

Specifies the *type* (or behavior) of this class.

Usage

To specify the type of class (if needed), use the following syntax:

```
Class MyApp.MyClass [ ClassType = classtype ]
{ //class members }
```

Where *classtype* is one of the following:

- `datatype` — the class is a data type class and is used to represent a literal value.
- `persistent` — the class represents data to be stored in the database.
- `serial` — the class represents data to be stored (in its serialized state) in another persistent object.
- `stream` — the class represents streaming data.
- `view` — the class is used to define an SQL view (see the ViewQuery keyword).
- `index` — the class is an index class, a specialized class that defines an index interface. For details, see %Library.FunctionalIndex in the class reference.
- An empty string, which indicates that this class has no specific type. Abstract classes typically do not specify a class type.

If this keyword is not specified, the class type is inherited from the primary superclass, if there is one.

Note that ClassType is specified for system classes such as %RegisteredObject, %SerialObject, %Persistent, and data type classes, so you do not generally need to specify this keyword if you subclass those classes.

Details

This keyword specifies how this class is to be used. The class compiler uses the ClassType keyword to determine how to compile a class. For example, if ClassType is `persistent`, the class compiler additionally invokes the storage compiler to generate persistence code for a class. Unless explicitly defined, the value of ClassType is either the default value or it is inherited from the primary superclass.

For persistent classes, an explicit ClassType statement is only required when standard persistence behavior is being overridden. If a class definition includes such a statement, it is either because a developer has specified it or because the class originated in code developed with an older version of InterSystems IRIS.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

If you omit this keyword, the class type is inherited from the primary superclass, if there is one.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ClientDataType

Specifies the client data type used when this data type is projected to client technologies. Applies only to data type classes.

Usage

To specify the client data type to use when this data type is projected to client technologies, use the following syntax:

```
Class MyApp.MyString [ ClientDataType = clienttype ]
{ //class members }
```

Where *clienttype* is one of the following:

- BIGINT
- BINARY
- BINARYSTREAM
- BOOLEAN
- CHARACTERSTREAM
- CURRENCY
- DATE
- DECIMAL
- DOUBLE
- FDATE
- FTIMESTAMP
- HANDLE
- INTEGER
- LIST
- LONGVARCHAR
- NUMERIC
- STATUS
- TIME
- TIMESTAMP
- VARCHAR (default)

Details

This keyword specifies the client data type used when this class is projected to client technologies. Every data type class must specify a client data type.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

The default client data type is VARCHAR.

See Also

- "Class Definitions" in this book
- "Defining Data Type Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ClientName

Enables you to override the default class name used in client projections of this class.

Usage

To override the default name for a class when it is projected to a client, use the following syntax:

```
Class MyApp.MyClass [ ClientName = clientclassname ]
{ //class members }
```

Where *clientclassname* is an unquoted string to be used as the client name, instead of the class name.

Details

This keyword lets you define an alternate name for a class when it is projected to a client (such as when using the InterSystems IRIS Java binding).

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the actual class name is used on the client.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

CompileAfter

Specifies that this class should be compiled after other (specified) classes.

Usage

To indicate that the class compiler should compile this class after other classes, use the following syntax:

```
Class MyApp.MyClass [ CompileAfter = classlist ]
{ //class members }
```

Where *classlist* is one of the following:

- A class name. For example:

  ```
  [ CompileAfter = MyApp.Class1 ]
  ```

- A comma-separated list of class names, enclosed in parentheses. For example:

  ```
  [ CompileAfter = (MyApp.Class1,MyApp.Class2,MyApp.Class3) ]
  ```

Details

This keyword specifies that the class compiler should compile this class after compiling the specified classes.

Typically this keyword is used when there is a dependency that the compiler cannot detect between classes such that one must be compiled after another.

This keyword affects only compilation order, not runtime behavior.

Note: The CompileAfter keyword does not ensure the specified classes are runnable before compiling this class. See the DependsOn keyword.

Also, the CompileAfter keyword affects only classes with common values for the System keyword.

Effect on Subclasses

This keyword is inherited from all superclasses. If the subclass specifies a value for the keyword, that value specifies *additional* classes that must be compiled before the subclass can be compiled.

Default

By default, this keyword is not specified.

See Also

- System keyword
- DependsOn keyword
- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

DdlAllowed

Specifies whether DDL statements can be used to alter or delete the class definition. Applies only to persistent classes.

Usage

To make it possible to modify a class via DDL, use the following syntax:

```
Class MyApp.Person Extends %Persistent [ DdlAllowed ]
{ //class members }
```

Otherwise, omit this keyword or use the following syntax:

```
Class MyApp.Person Extends %Persistent [ Not DdlAllowed ]
{ //class members }
```

Details

This keyword specifies whether DDL statements (such as DROP TABLE, ALTER TABLE, DROP INDEX, and so on) can be used to alter or delete the class definition.

Typically it is undesirable to enable SQL users to modify classes using DDL statements.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, DDL statements cannot be used to affect the class definition.

Note

If you create a class by executing a DDL CREATE TABLE statement, the DdlAllowed keyword will initially be set to true for that class.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

DependsOn

Specifies that this class should be compiled after the compiler has made other (specified) classes runnable.

Usage

To indicate that the class compiler should compile this class after other classes are runnable, use the following syntax:

```
Class MyApp.MyClass [ DependsOn = classlist ]
{ //class members }
```

Where *classlist* is one of the following:

- A class name. For example:

  ```
  [ DependsOn = MyApp.Class1 ]
  ```

- A comma-separated list of class names, enclosed in parentheses. For example:

  ```
  [ DependsOn = (MyApp.Class1,MyApp.Class2,...) ]
  ```

Details

This keyword specifies that the class compiler should compile this class after making the specified classes runnable.

This keyword is useful if compilation of this class uses these other classes in the method generator logic. It is also useful if the class contains initial expressions that invoke other classes.

This keyword affects only compilation order, not runtime behavior.

Note: The DependsOn keyword affects only classes with common values for the System keyword.

Also, if a class has `DependsOn=ClassA`, it is redundant for it to have `CompileAfter=ClassA` as well. See the CompileAfter keyword.

Effect on Subclasses

This keyword is inherited from all superclasses. If the subclass specifies a value for the keyword, that value specifies *additional* classes that must be runnable before the subclass can be compiled.

Default

By default, this keyword is not specified.

See Also

- System keyword
- CompileAfter keyword
- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Deprecated

Specifies that this class is deprecated. This keyword is ignored by the class compiler and by Studio, but is used by Atelier.

Usage

To mark a class as deprecated, use the following syntax:

```
Class MyApp.MyClass [ Deprecated ]
{ //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

See Also

- "Class Definitions" in this book

Final

Specifies whether this class is final (cannot have subclasses).

Usage

To specify that a class is final, use the following syntax:

```
Class MyApp.Exam As %Persistent [ Final ]  { //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If a class is *final*, it cannot have subclasses.

Also, if a class is final, the class compiler may take advantage of certain code generation optimizations (related to the fact that instances of a final class cannot be used polymorphically).

Default

If you omit this keyword, the class definition is not final.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

GeneratedBy

Indicates that this class was generated by code in another class and thus should not be edited.

Usage

The following syntax indicates that this class was generated by code in another class:

```
Class MyApp.MyClass [ GeneratedBy = MyApp.Generator.cls ] { //class members }
```

Where *MyApp.Generator* is a fully qualified class name.

Details

If this keyword is specified, Studio displays the class with a gray background to indicate that it should not be edited.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, Studio displays the class normally.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Hidden

Specifies whether this class is hidden (not listed in the class reference).

Usage

To make a class hidden, use the following syntax:

```
Class MyApp.Person [ Hidden ]  { //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If a class is *hidden*, it is not listed in the class reference nor in the Workspace window of the Studio Inspector. (It is still possible, however, to open the class in Studio, if you type its name in the **Open** dialog box.)

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the class is not hidden.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Inheritance

Specifies the inheritance order for the superclasses of this class.

Usage

To specify the inheritance order for the superclasses of this class, use the following syntax:

```
Class MyApp.MyClass Extends (MySuperClass1, MySuperClass2) [ Inheritance = inheritancedirection ] {
//class members }
```

Where *inheritancedirection* is `left` or `right`.

Or omit this keyword. In this case, InterSystems IRIS uses the default inheritance direction (`left`).

Details

The Inheritance keyword specifies inheritance order for a class with multiple inheritance. A value of `left` for *inheritancedirection* specifies left-to-right inheritance and a value of `right` specifies right-to-left inheritance.

For example, in the class definition in the synopsis, a value of `left` specifies that conflicting member definitions between MySuperClass1 and MySuperClass2 are resolved in favor of MySuperClass1; by contrast, a value of `right` specifies that conflicting member definitions between MySuperClass1 and MySuperClass2 are resolved in favor of MySuperClass2.

Important: The leftmost listed superclass is always the primary superclass, regardless of inheritance order.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the inheritance order is `left`.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Multiple Inheritance" in "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Language

Specifies the default language used to implement methods for this class.

Usage

To specify the default language used to implement methods in this class, use the following syntax:

```
Class MyApp.MyClass [ Language = language ]
{ //class members}
```

Where *language* is one of the following:

- `objectscript` — ObjectScript (the default)
- `tsql` — Transact-SQL

Or omit this keyword. In this case, InterSystems IRIS uses the default language (ObjectScript).

Details

This keyword specifies the default language used to implement methods for this class. Individual methods can override this value using the method's Language keyword.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the language is ObjectScript.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

LegacyInstanceContext

Specifies whether instance methods in this class can use the now-obsolete *%this* variable.

Usage

To enable instance methods in the class to use *%this*, use the following syntax:

```
Class MyApp.MyClass [ LegacyInstanceContext ]  { //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If this keyword is true, instance methods in this class can use the *%this* variable, which is now obsolete (replaced by $this). If this keyword is false, instance methods cannot refer to *%this*.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, instance methods *cannot* refer to *%this*.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

NoExtent

Specifies whether the compiler is prevented from generating an extent for this class (in the case where it would otherwise do so).

Usage

To prevent the compiler from generating an extent for this class (in cases where it would otherwise do so), use the following syntax:

```
Class MyApp.MyClass [ NoExtent ]  { //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If this keyword is true, the class has no extent. You cannot create instances of such a class. Frequently, such classes extend or override the standard persistent interface inherited from %Library.Persistent.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the class *can* have an extent.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

OdbcType

Specifies the type used when this data type is exposed via ODBC or JDBC. Every data type class must specify an ODBC type. This keyword applies only to data type classes.

Usage

To specify the type to use when this data type is projected via ODBC or JDBC, use the following syntax:

```
Class MyApp.MyString [ ClassType = DataType, OdbcType = odbctype ]  { //class members }
```

Where *odbctype* is one of the following:

- BIGINT
- BIT
- DATE
- DOUBLE
- INTEGER
- LONGVARBINARY
- LONGVARCHAR
- NUMERIC
- RESULTSET
- SMALLINT
- STRUCT
- TIME
- TIMESTAMP
- TINYINT
- VARBINARY
- VARCHAR (default)

Details

This keyword specifies the type used when exposed via ODBC or JDBC.

Every data type class must specify an ODBC type.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

If you omit this keyword, the ODBC type is VARCHAR.

See Also

- "Class Definitions" in this book
- "Defining Data Type Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Owner

Specifies the owner of this class and its corresponding table. Applies only to persistent classes.

Usage

To specify the owner of this class and its corresponding table, use the following syntax:

```
Class MyApp.Person Extends %Persistent [ Owner = "Jack" ]  { //class members }
```

Where *username* is an InterSystems IRIS user name.

Details

This keyword specifies the owner of the class and its corresponding table.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

If you omit this keyword, this class and its table are owned by the _SYSTEM user.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ProcedureBlock

Specifies whether each ObjectScript method in this class is a procedure block by default.

Usage

To cause the ObjectScript methods in this class to be procedure blocks by default, either *omit this keyword* or use the following syntax:

```
Class MyApp.MyClass [ ProcedureBlock ]  { //class members }
```

Otherwise, use the following syntax:

```
Class MyApp.MyClass [ Not ProcedureBlock ] { //class members }
```

Details

This keyword specifies whether ObjectScript methods in this class are procedure blocks by default. You can override this for an individual methods by setting the ProcedureBlock keyword for the method.

This keyword is ignored for methods written in other languages.

Within ObjectScript, methods can be implemented as procedure blocks or not. Procedure blocks enforce variable scoping: methods cannot see variables defined by their caller. New applications use procedure blocks; non-procedure blocks exist for backwards compatibility.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, each ObjectScript method in this class is a procedure block (unless you override that for individual methods).

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

PropertyClass

Adds property parameters to this class.

Usage

To add property parameters to this class, use the following syntax:

```
Class PropClass.MyClass Extends %RegisteredObject [ PropertyClass = PropClass.MyPropertyClass ] {
//class members }
```

Where *propertyclasslist* is one of the following:

- A full class name, including all packages. For example:

  ```
  [ PropertyClass = PropClass.MyPropertyClass ]
  ```

- A comma-separated list of class names, enclosed in parentheses.

Details

If you need to add custom property parameters, do the following:

1. Define and compile a class that defines one or more class parameters. For example:

   ```
   Class PropClass.MyPropertyClass
   {

   Parameter MYPARM As %String = "XYZ";

   }
   ```

   These class parameters become property parameters in the next step.

2. In the class that defines the properties, specify the PropertyClass keyword.

Effect on Subclasses

Subclasses inherit the custom behavior added by this keyword. If the subclass specifies a value for the keyword, that value specifies an *additional* class or classes that specify parameters for properties of this class.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ServerOnly

Specifies whether this class is projected to Java clients.

Usage

To override the default way that the class would be projected to Java clients, use the following syntax:

```
Class Sample.NewClass1  [ ServerOnly = serveronlyvalue ] { //class members }
```

Where *serveronlyvalue* is one of the following:

- `0` means that this class can be projected.
- `1` means that this class will not be projected.

Details

If this keyword is 1, the class will not be projected to a Java client. If this keyword is 0, the class will be projected.

Effect on Subclasses

This keyword is not inherited.

Default

If this keyword is omitted, this class is projected if it is *not* a stub (but is not projected if it is a stub).

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SoapBindingStyle

Specifies the binding style or SOAP invocation mechanism used by any web methods defined in this class. Applies only in a class that is defined as a web service or web client.

Usage

To specify the binding style used by any web methods defined in this class, use the following syntax:

```
Class MyApp.MyClass [ SoapBindingStyle = soapbindingstyle ]  { //class members }
```

Where *soapbindingstyle* is one of the following:

- `document` (default) — Web methods in this class use document-style binding by default.

  With this binding style, the SOAP messages are formatted as documents and typically have only one part.

  In the SOAP messages, the `<Body>` element typically contains a single child element. Each child of the `<Body>` element corresponds to a message part.

- `rpc` — Web methods in this class use RPC (remote procedure call)-style binding by default.

  With this binding style, the SOAP messages are formatted as messages with multiple parts.

  In the SOAP messages, the `<Body>` element contains a single child element whose name is taken from the corresponding operation name. This element is a generated wrapper element, and it contains one child element for each argument in the argument list of the method.

If SoapBindingStyle is `document` and if *ARGUMENTSTYLE* is `message`, then the message style is very similar to RPC; see *Creating Web Services and Web Clients*.

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword lets you specify the default binding style used by any web methods defined in this class. It affects the format of the SOAP body (but not any SOAP headers).

You can override the binding style for individual methods, by using the SoapBindingStyle method keyword or the SoapBindingStyle query keyword.

Effect on Subclasses

This keyword is not inherited.

Default

The default value is `document`. (Chapter 7 of the SOAP standard v1.1, "Using SOAP for RPC," specifies that web methods should use RPC-style binding. However, most SOAP clients, including .NET, use document-style binding.)

Relationship to WSDL

The SoapBindingStyle class keyword specifies the value of the `style` attribute of `<soap:binding>` element within the `<binding>` section of the WSDL. For example, if SoapBindingStyle is `document`, the WSDL could look as follows:

```
...
<binding ...>
    <soap:binding ... style="document"/>
    <operation ...>
        <soap:operation ... style="document"/>
...
```

As shown here, the SoapBindingStyle class keyword also specifies the default value of the `style` attribute of the `<soap:operation>` element, within the `<binding>` section of the WSDL; this attribute is further controlled by the SoapBindingStyle method keyword.

In contrast, if SoapBindingStyle is `rpc`, the WSDL could instead be as follows:

```
...
<binding ...>
    <soap:binding ... style="rpc"/>
    <operation ...>
        <soap:operation ... style="rpc"/>
...
```

The binding style also affects the `<message>` elements, as follows:

- If the binding style is `document`, a message has only one part by default. For example:

  ```
  <message name="AddSoapIn">
      <part name="parameters" .../>
  </message>
  ```

  If the *ARGUMENTSTYLE* parameter is `message`, then a message can have multiple parts. For example:

  ```
  <message name="AddSoapIn">
     <part name="a" .../>
     <part name="b" .../>
  </message>
  ```

- If the binding style is `rpc`, a message can have multiple parts. For example:

  ```
  <message name="AddSoapIn">
      <part name="a" .../>
      <part name="b" .../>
  </message>
  ```

Effect on SOAP Messages

The primary effect on SOAP messages is to control whether the SOAP body can contain multiple subelements.

For a web method that uses a RPC-style binding and encoded-style messages, the following shows an example of the body of a possible request message:

```
<SOAP-ENV:Body SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
   <types:Add>
      <a href="#id1" /><b href="#id2" />
   </types:Add>
   <types:ComplexNumber id="id1" xsi:type="types:ComplexNumber">
      <Real xsi:type="s:double">10</Real>
      <Imaginary xsi:type="s:double">5</Imaginary>
   </types:ComplexNumber>
   <types:ComplexNumber id="id2" xsi:type="types:ComplexNumber">
      <Real xsi:type="s:double">17</Real>
      <Imaginary xsi:type="s:double">2</Imaginary>
   </types:ComplexNumber>
</SOAP-ENV:Body>
```

In contrast, the following shows an example of the body of a possible request message for a web method that uses literal binding and encoded-style messages:

```
<SOAP-ENV:Body>
   <tns:Add>
      <tns:a xsi:type="tns:ComplexNumber">
         <Real xsi:type="s:double">10</Real>
         <Imaginary xsi:type="s:double">5</Imaginary>
      </tns:a>
      <tns:b xsi:type="tns:ComplexNumber">
         <Real xsi:type="s:double">17</Real>
         <Imaginary xsi:type="s:double">2</Imaginary>
      </tns:b>
      </tns:Add>
</SOAP-ENV:Body>
```

In this case, the SOAP body has a single subelement.

Use With %XML.DataSet

For objects of type %XML.DataSet, not all permutations of the SoapBindingStyle and SoapBodyUse keywords are permitted, as the following table summarizes:

	SoapBodyUse=literal (default)	*SoapBodyUse=encoded*
SoapBindingStyle=document(default)	supported	not supported
SoapBindingStyle=rpc	supported	supported

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapBodyUse

Specifies the encoding for any web methods defined in this class. This keyword applies only to web service and web client classes.

Usage

To specify the encoding used by the inputs and outputs of the web methods of this class, use the following syntax:

```
Class MyApp.MyClass [ SoapBodyUse = soapbodyuse ]  { //class members }
```

Where *soapbodyuse* is one of the following:

- `literal` (default) — Web methods in this class use literal data by default. That is, the XML within the `<Body>` of the SOAP message exactly matches the schema given in the WSDL.
- `encoded` — Web methods in this class use SOAP-encoded data by default. That is, the XML within the `<Body>` of the SOAP message uses SOAP encoding as appropriate for the SOAP version being used, as required by the following specifications:
  - SOAP 1.1 (http://www.w3.org/TR/2000/NOTE-SOAP-20000508/)
  - SOAP 1.2 (http://www.w3.org/TR/soap12-part2/)

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword specifies the default encoding used by any web methods defined in this class. It also controls the default values for the *ELEMENTQUALIFIED* and *XMLELEMENT* parameters for this class, as discussed in a subsection of this topic.

You can override this keyword for individual methods, by using the SoapBodyUse method keyword or the SoapBodyUse query keyword.

Effect on Subclasses

This keyword is not inherited.

Default

The default value is `literal`. (The SOAP standard v1.1 (Chapter 5) specifies that web methods should use SOAP encoding. However, most SOAP clients, including .NET, use literal style.)

Relationship to WSDL

The SoapBodyUse keyword specifies the value of the `use` attribute of the `<soap:body>` element within the `<binding>` section of the WSDL. For example, if SoapBodyUse is `literal`, the WSDL could look as follows:

```
...
<binding name="MyServiceNameSoap"
 ...
    <soap:binding ...
    <operation name="Add">
        <soap:operation ...>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
...
```

In contrast, if SoapBodyUse is `encoded`, the WSDL could instead be as follows:

```
...
<binding name="MyServiceNameSoap" ...
    <soap:binding ...
    <operation name="Add">
        <soap:operation .../>
        <input>
            <soap:body use="encoded" namespace="http://www.mynamespace.org"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
            <soap:body use="encoded" namespace="http://www.mynamespace.org"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>

    </operation>
</binding>
...
```

For SOAP 1.2, the `encodingStyle` attribute is as follows instead:

```
encodingStyle="http://www.w3.org/2003/05/soap-encoding"
```

The SoapBodyUse keyword also determines the contents of a `<part>` element of a `<message>` element for each web method:

- If SoapBodyUse is `literal`, each `<part>` element includes an `element` attribute. For example:

  ```
  <part name="parameters" element="s0:Add"/>
  ```

  For another example:

  ```
  <part name="b" element="s0:b"/>
  ```

- If SoapBodyUse is `encoded`, each `<part>` element includes a `type` attribute rather than an `element` attribute. For example:

  ```
  <part name="a" type="s0:ComplexNumber"/>
  ```

Note that SoapBodyUse also controls the default values for the *ELEMENTQUALIFIED* and *XMLELEMENT* parameters, which also affect the WSDL.

Effect on SOAP Messages

For a web method that uses a document-style message, the web service sends a response message like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'>
   <SOAP-ENV:Body>
      <AddResponse ...>
...
```

In contrast, for a web service that uses an encoded-style message, the response message would be as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'
                   xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
                   xmlns:tns='http://www.mynamespace.org'
                   xmlns:types='http://www.mynamespace.org'>
   <SOAP-ENV:Body SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
      <types:AddResponse>
...
```

Effect on Default for Parameters of the Web Service or Web Client

The default value for the *ELEMENTQUALIFIED* parameter depends on the SoapBodyUse keyword:

Value of SoapBodyUse	Default for ELEMENTQUALIFIED	Notes
`literal`	1	elementFormDefault="qualified"
`encoded`	0	elementFormDefault="unqualified"

The default value for the *XMLELEMENT* parameter also depends on the SoapBodyUse keyword:

Value of SoapBodyUse	Default for XMLELEMENT	Notes
`literal`	1	Message parts have the `element` attribute
`encoded`	0	Message parts have the `type` attribute

For details on the *ELEMENTQUALIFIED* and *XMLELEMENT* parameters, see *Projecting Objects to XML*.

Use with %XML.DataSet

For objects of type %XML.DataSet, not all permutations of the SoapBindingStyle and SoapBodyUse keywords are permitted. See the entry for the SoapBindingStyle class keyword.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SqlCategory

Specifies the type to use for calculations in SQL. Applies only to data type classes.

Usage

To specify the type to use for calculations in SQL, use the following syntax:

```
Class MyApp.MyString [ ClassType = DataType, SQLCategory = STRING ]  { //class members }
```

Where *sqlcategory* is one of the following:

- `DATE`
- `DOUBLE`
- `FMDATE`
- `FMTIMESTAMP`
- `INTEGER`
- `MVDATE`
- `NAME`
- `NUMERIC`
- `STRING` (default)
- `TIME`
- `TIMESTAMP`

Details

This keyword specifies the type to use for this class in SQL calculations.

Every data type class must specify an SQL category.

When creating a new data type class, use the SQL category value that most closely matches the data type you are creating, or, better still, subclass an existing data type class and inherit its SQL category.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

The default SQL category is `STRING`.

See Also

- "Class Definitions" in this book
- "Defining Data Type Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlRowIdName

Overrides the default SQL field name for the ID column for this class. Applies only to persistent classes.

Usage

To override the default SQL field name for the ID column of this class, use the following syntax:

```
Class MyApp.MyClass [ SqlRowIdName = MyId ]  { //class members }
```

Where *MyId* is an SQL identifier.

Details

This keyword overrides the default SQL field name used for the ID column.

When a persistent class is projected as an SQL table, the Object Identity value for each object is projected as an SQL column — the Row ID column. By default, the Row ID column is called ID. If the class has another field named ID, then ID1 is used (and so on). The SqlRowIdName keyword lets you set the name of the Row ID column directly.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

If you omit this keyword, the SQL field name for the ID column for this class is `ID`

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlRowIdPrivate

Specifies whether the ID column for this class is a hidden field when projected to ODBC and JDBC. Applies only to persistent classes.

Usage

To hide the ID column when projecting the table to ODBC and JDBC, use the following syntax:

```
Class MyApp.MyClass [ SqlRowIdPrivate ]  { //class members }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If this keyword is true, the ID column is a hidden field when the table is projected to ODBC and JDBC.

When a persistent class is projected as an SQL table, the Object Identity value for each object is projected as an SQL column — the Row ID column. The SqlRowIdPrivate keyword lets you specify whether this Row ID column should be "hidden" from ODBC and JDBC-based queries. If the row ID column is hidden:

- It is not reported as a column by the various catalog queries
- It is not included in the `SELECT *` query.

An ODBC or JDBC client *can* select this column if the query explicitly lists the column within the SELECT clause. (Note that by definition, you cannot use a Row ID column in an UPDATE or INSERT statement because the value of the Row ID cannot be modified or directly set).

Typically you use this keyword for cases where you are dealing with legacy relational data and do not want the Row ID column to be seen by reporting tools.

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the ID column is visible normally when the table is projected to ODBC and JDBC.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlTableName

Specifies the name of the SQL table to which this class is projected. Applies only to persistent classes.

Usage

To override the default name of the SQL table to which this class is projected, use the following syntax:

```
Class MyApp.Person Extends %Persistent [ SqlTableName = DBTable ]  { //class members }
```

Where *DBTable* is a valid SQL identifier.

Details

This keyword specifies the name of the SQL table to which this class is projected. By default, the SQL table name is the same as the class name.

Typically you use this keyword when the class name is a SQL reserved word (not uncommon) or if you want the SQL table to contain characters not supported by class names (such as the "_" character).

Effect on Subclasses

This keyword is not inherited.

Default

If you omit this keyword, the class name is used as the SQL table name.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

StorageStrategy

Specifies which storage definition controls persistence for this class. Applies only to persistent and serial classes.

Usage

To specify which storage definition the class uses, use syntax like the following:

```
Class MyApp.MyClass Extends %Persistent [ StorageStrategy = MyStorage ]
{ //class members }
```

Where *MyStorage* is the name of a storage definition in this class.

Details

This keyword specifies which storage definition is used to define the storage structure used by this class.

Typically you do not worry about this keyword or about storage structures; the class compiler will automatically define a storage structure named "Default" and maintain it (add new fields as appropriate) for you. It is possible to create more than one storage definition for a class. In this case, this keyword is used to specify which storage definition the class compiler should use.

Effect on Subclasses

This keyword is inherited from the primary superclass. The subclass can override the value of the keyword.

Default

If you omit this keyword, the persistence for this class is defined by the default storage definition, named `Default`.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

System

Influences the compilation order for this class.

Usage

To influence the compilation order for a class, use syntax like the following:

```
Class MyApp.Person Extends %Persistent [ System = n ]
{ //class members }
```

Where *n* is an integer ranging from 0 to 4, where classes with the lower positive values are compiled before classes with higher positive values. Classes with values of 0 (zero) are compiled last.

Details

This keyword establishes groups of classes, each associated with a different value and priority, where the full class compilation process occurs for each priority level before moving on to the subsequent priority level. From highest priority to lowest priority, the levels are:

- 1
- 2
- 3
- 4
- 0 (the default)

Class compilation has two steps:

1. Resolving the globals.
2. Compiling the routines.

All classes with the same value of the System keyword have their globals resolved before routine compilation. With classes of varying levels, those of higher priority have both resolved globals and compiled routines before those of lower priority have their globals resolved.

The CompileAfter and DependsOn keywords work within the classes with a common System value to determine the order of global resolution. Once all classes with a common System value have had their globals resolved, then routine compilation proceeds for all of them.

Hence, if class B needs to run a method of class A in class B's method generator (that is, during compilation of B), then A must have a higher priority than B. This means that the value for A's System keyword must be a non-zero integer that is lower than the value for B. To obtain this behavior, CompileAfter or DependsOn do not work.

Effect on Subclasses

This keyword is not inherited.

Default

The default value is 0 (zero).

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*

- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ViewQuery

Specifies the SQL query for this class. Applies only to view definition classes.

Usage

To specify the SQL query for this class, use the following syntax:

```
ViewQuery = { statement }
```

Where *statement* is an SQL SELECT statement, enclosed in curly braces.

Description

When you define an SQL view (using the DDL CREATE VIEW statement or using the Management Portal), the system automatically creates a class definition to hold the view definition. For this class definition, ClassType is `view`, and ViewQuery equals the SQL statement on which the view is based.

This mechanism is internal; users are not expected to create view classes nor to modify the ViewQuery keyword. Instead, use the normal mechanisms (DDL or the Management Portal) for managing views.

This keyword is ignored for all non-view classes.

Default

The default value is an empty string.

See Also

- "Class Definitions" in this book
- "Defining and Compiling Classes" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Foreign Key Keywords

This reference describes the keywords that apply to a foreign key, which you can define within persistent classes. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on foreign key definitions, see “Foreign Key Definitions.”

Internal

Specifies whether this foreign key definition is internal (not displayed in the class documentation).

Usage

To mark this foreign key definition as internal, use the following syntax:

```
ForeignKey keyname(key_props) References pkg.class(ref_index) [ Internal ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this foreign key definition is displayed in the class documentation.

See Also

- "Foreign Key Definitions" in this book
- "Using Foreign Keys" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

NoCheck

Specifies whether InterSystems IRIS should check this foreign key constraint.

Usage

To prevent InterSystems IRIS from checking the constraint defined by this foreign key, use the following syntax:

```
ForeignKey keyname(key_props) References pkg.class(ref_index) [ NoCheck ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

The NoCheck keyword suppresses the checking of the foreign key constraint (in other words, it specifies that the foreign key constraint never be checked).

Default

If you omit this keyword, InterSystems IRIS does check the foreign key constraint.

See Also

- "Foreign Key Definitions" in this book
- "Using Foreign Keys" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

OnDelete

Specifies the action that this foreign key should cause in the current table when a record is deleted in the foreign table.

Usage

To specify what happens in the current table when a record is deleted in the foreign table, use the following syntax:

```
ForeignKey keyname(key_props) References pkg.class(ref_index) [ OnDelete = ondelete ];
```

Where *ondelete* is one of the following:

- `noaction` (default) — When an attempt is made to delete a record in the foreign table, the attempt fails.
- `cascade` — When a record is deleted in the foreign table, the referencing record in this table is also deleted.
- `setdefault` — When a record is deleted in the foreign table, the referencing record in this table is set to its default value.
- `setnull` — When a record is deleted in the foreign table, the referencing record in this table is set to null.

Description

This keyword defines the referential action that occurs when a record is deleted from the foreign table.

Default

The default is `noaction`.

See Also

- "Foreign Key Definitions" in this book
- "Using Foreign Keys" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

OnUpdate

Specifies the action that this foreign key should cause in the current table when a record is updated in the foreign table.

Usage

To specify what happens in the current table when a record is updated in the foreign table, use the following syntax:

```
ForeignKey keyname(key_props) References pkg.class(ref_index) [ OnUpdate = onupdate ];
```

Where *onupdate* is one of the following:

- `noaction` (default)
- `cascade`
- `setdefault`
- `setnull`

Details

This keyword defines the referential action that occurs when a key value is updated in the foreign table.

Default

The default is `noaction`.

See Also

- "Foreign Key Definitions" in this book
- "Using Foreign Keys" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlName

Specifies an SQL alias for the foreign key.

Usage

To override the default SQL name for this foreign key, use the following syntax:

```
ForeignKey keyname(key_props) References pkg.class(ref_index) [ SqlName = alternate_name ];
```

Where *alternate_name* is an SQL identifier.

Details

This keyword lets you define an alternate name for this foreign key when referred to via SQL.

Default

If you omit this keyword, the SQL name for the foreign key is *keyname* as specified in the foreign key definition.

See Also

- "Foreign Key Definitions" in this book
- "Using Foreign Keys" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Index Keywords

This reference describes the keywords that apply to an index, which you can define in persistent classes. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on index definitions, see "Index Definitions."

Abstract

Specifies that an index is abstract.

Usage

To specify that an index is abstract, use the following syntax:

```
Index MyIndex [ Abstract ];
```

Note: When you create a sharded table, an abstract shard key index is generated automatically and there is no need to define one.

Details

Abstract indices are intended for use only with sharded tables. They contain no data and thus have no storage (no index global). A sharded table has exactly one abstract index, called the *shard key index*. The purpose of a shard key index is to serve as the key that determines the shard in which a row resides.

If an index is defined as abstract, the index is not accessible or usable via methods or via SQL. If you also try to mark this index as unique or try to use it within a primary key, those constraints are ignored.

You cannot define an IdKey index as abstract. If you attempt to do so, a class compilation error will result.

You can use this keyword on an existing index to make it abstract. This will not delete any existing data in the index.

Default

The default for the Abstract keyword is false.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Condition

Defines a conditional index and specifies the condition that must be met for a record to be included in the index.

Usage

This keyword is for migrating existing applications to InterSystems SQL and is not for use in new applications.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

CoshardWith

Specifies the name of the class with which this class is cosharded.

Usage

This keyword has been defined, but it will not be functional until a future release. Sharded tables can currently be created and accessed using SQL only.

See Also

- "Horizontally Scaling InterSystems IRIS for Data Volume with Sharding" in the *Scalability Guide*
- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Data

Specifies a list of properties whose values are to be stored within this index.

Usage

To store values of properties within an index, use the following syntax:

```
Index name On property_expression_list [ Data = stored_property_list ];
```

Where *stored_property_list* is either a single property name or a comma-separated list of properties, enclosed in parentheses.

Details

This keyword specifies a list of properties whose values are to be stored within this index.

You cannot use this keyword with a bitmap index.

Refer to the documentation on indices for more details.

Default

If you omit this keyword, values of properties are not stored within the index.

Example

```
Index NameIDX On Name [ Data = Name ];
Index ZipIDX On ZipCode [ Data = (City,State) ];
```

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Extent

Defines an extent index.

Usage

To specify that this is an extent index, use the following syntax:

```
Index State [ Extent ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Note: If you are using bitmap indices, then an extent index is automatically maintained and there is no need to define one.

Details

An extent index is used to keep track of which object instances belong to a subclass.

Default

The default for the Extent keyword is false.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

IdKey

Specifies whether this index defines the Object Identity values for the table.

Usage

To specify that the Object Identity values for this table should be formed from the property or properties on which this index is based, use the following syntax:

```
Index name On property_expression_list [ IdKey ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that the property or properties on which this index is based will be used to form the Object Identity value for this object.

Important: There must not be a sequential pair of vertical bars (`||`) within the values of any property used by an IDKEY index, unless that property is a valid reference to an instance of a persistent class. This restriction is imposed by the way in which the InterSystems SQL mechanism works. The use of `||` in IDKey properties can result in unpredictable behavior.

The Object Identity value is used to uniquely locate persistent object instances. Once an object using IdKey has been saved, you cannot modify the values of any of the properties that make up the IdKey.

An IdKey index also behaves like a unique index. That is, for the property (or the combination of properties) that you use in this index, InterSystems IRIS enforces uniqueness. It is permitted, but redundant, to specify the Unique keyword as true in this index definition.

Default

The default for the IdKey keyword is false.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this index definition is internal (not displayed in the class documentation).

Usage

To specify that this index definition is internal, use the following syntax:

```
Index name On property_expression_list [ Internal ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this index is displayed in the class documentation.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

PrimaryKey

Specifies whether this index defines the primary key for the table.

Usage

To specify that the primary key for this table is formed by the properties on which this index is based, use the following syntax:

```
Index name On property_expression_list [ PrimaryKey ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that this index should be reported, via SQL, as being the *primary key* for this class (table).

A PrimaryKey index also behaves like a unique index. That is, for the property (or the combination of properties) that you use in this index, InterSystems IRIS enforces uniqueness. It is permitted, but redundant, to specify the Unique keyword as true in this index definition.

Example

```
Index EmpIDX On EmployeeID [ PrimaryKey] ;
```

Default

If you omit this keyword, the primary key for this table is *not* formed by the properties on which this index is based.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlName

Specifies an SQL alias for the index.

Usage

To override the default name for this index when referred to via SQL, use the following syntax:

```
Index name On property_expression_list [ SqlName = sqlindexname];
```

Where *sqlindexname* is an SQL identifier.

Details

This keyword lets you define an alternate name for this index when referred to via SQL.

Default

If you omit this keyword, the SQL name of the index is *indexname* as given in the index definition.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Type

Specifies the type of index.

Usage

To specify the type of the index, use the following syntax:

```
Index name On property_expression_list [ Type = indextype ];
```

Where *indextype* is one of the following:

- `bitmap` — a bitmap index
- `bitslice` — a bitslice index
- `index` — a standard index (default)
- `key` — deprecated

Details

This keyword specifies the type of the index, specifically whether the index is implemented as a bitmap index or a standard (regular, non-bitmap) index.

A bitmap index cannot be marked as unique.

Default

If you omit this keyword, the index is a standard index.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Unique

Specifies whether the index should enforce uniqueness.

Usage

To specify that InterSystems IRIS should enforce uniqueness for the properties on which this index is based, use the following syntax:

```
Index name On property_expression_list [ Unique ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

The Unique keyword, if present, indicates that this is a *unique* index.

The property (or properties) indexed by a unique index are constrained to have unique values (that is, no two instances can have the same collated value) within the extent (set of all objects) of the class (table) that defines the index.

A unique index cannot also be a bitmap index.

Example

```
Index SSNIdx On SSN [ Unique ] ;
```

Default

If you omit this keyword, InterSystems IRIS does *not* enforce uniqueness for the properties on which this index is based.

See Also

- "Index Definitions" in this book
- "Defining and Building Indices" in the *SQL Optimization Guide*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Method Keywords

This reference describes the keywords that apply to a method. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on method definitions, see “Method Definitions.”

Abstract

Specifies whether this is an *abstract* method.

Usage

To specify that this method is abstract, use the following syntax:

```
Method name(formal_spec) As returnclass [ Abstract ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

An *abstract* method has no implementation and has no executable code generated for it. Abstract methods exist solely for the purpose of defining a method signature (or interface) that can be overridden and implemented within one or more subclasses. Some examples of abstract methods are the various callback methods defined, but not implemented by, in the InterSystems IRIS class library.

Default

If you omit this keyword, the method is not abstract.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ClientName

Overrides the default name for the method in client projections.

Usage

To override the default name for this method when the class is projected to a client language, use the following syntax:

```
Method name(formal_spec) As returnclass [ ClientName = clientname ]
{    //implementation }
```

Where *clientname* is the name to use in the client language.

Details

This keyword lets you define an alternate name for a method when it is projected to a client language. This is especially useful if the method name contains characters that are not allowed in the client language.

Default

If you omit this keyword, the method name is used as the client name.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

CodeMode

Specifies how this method is implemented.

Usage

To specify how the method is implemented, use the following syntax:

```
Method name(formal_spec) As returnclass [ CodeMode=codemode ]
{    //implementation }
```

Where *codemode* is one of the following:

- `call` — this method is an alias for a routine call (used for wrapping legacy code).
- `code` (default) — this method is implemented as lines of code.
- `expression` — this method is implemented as an expression.
- `objectgenerator` — this method is a method generator.

Note: There is an older value for this keyword (`generator`), which indicates that the older, non-object-based method generator should be used. This is only present for compatibility with older versions. Newer applications should use `objectgenerator`.

Details

This keyword specifies how a given method is implemented.

Typically, a method is implemented using one or more lines of code. This is indicated by the default CodeMode value of `code`. In this case, the method implementation is one or more lines of code.

Certain simple methods can be implemented as expression methods; in certain cases the class compiler may replace a call to this method with inline code containing the expression. In this case, the method implementation is a simple expression (with no Quit or **Return** statement).

A call method is a wrapper for a routine. In this case, the method implementation is the name of a routine and tag name.

Method generators are programs, invoked by the class compiler when a class is compiled, that generate the actual implementation for the given method. In this case, the method implementation is the code for the method generator. See “Defining Method and Trigger Generators” in Defining and Using Classes

Default

The default value for the CodeMode keyword is `code`.

Examples

```
/// An expression method
Method Double(val As %Integer) As %Integer [ CodeMode = expression ]
{
  val * 2
}

/// A Method generator
Method GetClassName() As %String [ CodeMode = objectgenerator ]
{
   Do %code.WriteLine(" Quit """ _ %class.Name _ """")
   Quit $$$OK
}
```

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Deprecated

Specifies that this method is deprecated. This keyword is ignored by the class compiler and by Studio, but is used by Atelier.

Usage

To specify that this method is deprecated, use the following syntax:

```
Method name(formal_spec) As returnclass [ Deprecated ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

See Also

- "Method Definitions" in this book

ExternalProcName

Specifies the name of this method when it is used as a stored procedure in a foreign database. Applies only if the method is projected as a stored procedure.

Usage

To override the default name of the method when it is used as a stored procedure in a foreign database, use the following syntax:

```
ClassMethod name(formal_spec) As returnclass [ SqlProc, ExternalProcName = MyProcedure ]
{    //implementation }
```

Where *MyProcedure* is an unquoted string.

Details

This keyword lets you define the name to use for this method when it is used as a stored procedure in a foreign database.

Default

If you omit this keyword, the method name is used as the stored procedure name.

See Also

- "Method Definitions" in this book
- SqlProc keyword
- "Defining Stored Procedures" in *Using InterSystems SQL*
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Final

Specifies whether this method is final (cannot be overridden in subclasses).

Usage

To specify that a method is final, use the following syntax:

```
Method name(formal_spec) As returnclass [ Final ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A class member that is marked as *final* cannot be overridden in subclasses.

Default

If you omit this keyword, the method is not final.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ForceGenerate

Specifies whether the method should be compiled in every subclass. Applies only if the method is a method generator.

Usage

To specify that the method (a method generator) should be compiled in every subclass, use the following syntax:

```
Method name(formal_spec) As returnclass [ CodeMode = ObjectGenerator, ForceGenerate ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

In the case of a method generator method, specifies that the method should be compiled in every subclass. This keyword is useful when you need to ensure that each subclass has its version of the method. InterSystems IRIS does not recompile a method in a subclass if the generated code looks the same as the superclass generated code. This logic does not consider whether the include files are the same for both classes. If the method uses a macro that is defined in an include file and if the subclass uses a different include file, InterSystems IRIS would not recompile the method in the subclass. In such a scenario, specify ForceGenerate for the method generator.

Default

If you omit this keyword, the method is not compiled in every subclass.

See Also

- "Method Definitions" in this book
- CodeMode keyword
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

GenerateAfter

Specifies when to generate this method. Applies only if the method is a method generator.

Usage

To specify that the generator for this method should be invoked after other methods are generated, use the following syntax:

```
Method name(formal_spec) As returnclass [ CodeMode = ObjectGenerator, GenerateAfter = methodlist ]
{    //implementation }
```

Where *methodlist* is either a single method name or a comma-separated list of method names, enclosed in parentheses.

Details

In the case of a method generator method, specifies that the generator should be invoked after the listed methods are generated. This keyword is useful when you need to control the order in which your method generators are invoked.

See Also

- "Method Definitions" in this book
- CodeMode keyword
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this method definition is internal (not displayed in the class documentation).

Usage

To specify that this method is internal, use the following syntax:

```
Method name(formal_spec) As returnclass [ Internal ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this method is displayed in the class documentation.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Language

Specifies the language used to implement this method.

Usage

To specify the language used to implement the method, use the following syntax:

```
Method name(formal_spec) As returnclass [ Language = language ]
{    //implementation }
```

Where *language* is one of the following:

- `objectscript` (the default) — ObjectScript
- `ispl` — Informix Stored Procedure Language
- `tsql` — Transact-SQL

Details

This keyword specifies the language used to implement this method.

The values `ispl` and `tsql` are only supported for class methods.

If you specify a value of `ispl`, the body of the method is limited to a single **CREATE PROCEDURE** statement.

Default

If you omit this keyword, the language specified by the class-level Language keyword is used.

Note: You cannot specify `Language = ispl` at the class level; you can only use this value for methods.

Examples

```
Class User.Person Extends %Persistent
{

Property Name As %String;

Property Gender As %String;

/// An ObjectScript instance method that writes the name and gender of a person
Method Print() As %Status [ Language = objectscript ]
{
   write !, ..Name, " is a ", ..Gender
}

/// A TSQL class method that inserts a row into the Person table
ClassMethod TSQLTest() As %Status [ Language = tsql ]
{
   INSERT INTO Person (Name, Gender) VALUES ('Manon', 'Female')
}

/// An ISPL class method that creates an stored procedure named IsplSp
ClassMethod ISPLTest() As %Status [ Language = ispl ]
{
   CREATE PROCEDURE IsplSp()
      INSERT INTO Person (Name, Gender) VALUES ('Nikolai', 'Male')
   END PROCEDURE
}

}
```

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*

- “Defining Method and Trigger Generators” in *Defining and Using Classes*
- “Introduction to Compiler Keywords” in *Defining and Using Classes*

NotInheritable

Specifies whether this method can be inherited in subclasses.

Usage

To specify that this method cannot be inherited in subclasses, use the following syntax:

```
Method name(formal_spec) As returnclass [ NotInheritable ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that this method cannot be inherited in subclasses.

Important: While having a member not be inheritable by its subclasses can be very useful in some cases, the keyword should be used rarely and judiciously, as it breaks the inheritance contract.

Default

If you omit this keyword, this method is inheritable.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

PlaceAfter

Specifies the order of this method, relative to other methods, in the routine that is generated for the class.

Usage

To specify that the class compiler should place this method after the listed methods in the routine it creates for the class, use the following syntax:

```
Method name(formal_spec) As returnclass [ PlaceAfter = methodlist ]
{    //implementation }
```

Where *methodlist* is either a single method name or a comma-separated list of method names, enclosed in parentheses.

Details

This keyword specifies that the class compiler should place this method after the listed methods in the routine it creates for the class. This keyword is for rare cases where you need to control the order in which the class compiler generates code for your method.

Default

If you omit this keyword, the class compiler uses its normal logic to determine the order of the methods in the routine that it generates.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Private

Specifies whether this method is private (can be invoked only by methods of this class or its subclasses).

Usage

To specify that the method is private, use the following syntax:

```
Method name(formal_spec) As returnclass [ Private ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Private class members can be used only by other members of the same class (or its subclasses). Note that other languages often use the word *protected* to describe this kind of visibility and use the word *private* to mean invisibility from subclasses.

This keyword is inherited but you can change its value in subclasses.

Default

If you omit this keyword, this method is not private.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ProcedureBlock

Specifies whether this method is a procedure block. Applies only if the method is written in ObjectScript.

Usage

The class definition specifies whether methods in the class are procedure blocks by default. To override that default, and specify that a given method is a procedure block, use the following syntax:

```
Method name(formal_spec) As returnclass [ ProcedureBlock ]
{    //implementation }
```

Or (equivalently):

```
Method name(formal_spec) As returnclass [ ProcedureBlock=1 ]
{    //implementation }
```

Otherwise, to specify that a given method is not a procedure block, use the following syntax:

```
Method name(formal_spec) As returnclass [ ProcedureBlock=0 ]
{    //implementation }
```

Details

This keyword specifies that an ObjectScript method is a procedure block.

Within ObjectScript, methods can be implemented as procedure blocks or not. Procedure blocks enforce variable scoping: methods cannot see variables defined by their caller. New applications use procedure blocks; non-procedure blocks exist for backwards compatibility.

Default

If you omit this keyword, the value of the class-level ProcedureBlock keyword is used.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

PublicList

Specifies the public variables for this method. Applies only if the method is written in ObjectScript and is a procedure block.

Usage

To specify the list of public variables for the method, use the following syntax:

```
Method name(formal_spec) As returnclass [ PublicList = variablelist ]
{    //implementation }
```

Where *publiclist* is either a single variable name or a comma-separated list of variable names, enclosed in parentheses.

Details

This keyword is used only if method is written in ObjectScript and is a procedure block. In ObjectScript, the public list specifies a list of variables that are scoped as *public* variables. Public variables are visible to any methods invoked from the method defining the public list.

Default

If you omit this keyword, the method has no public variables.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ReturnResultsets

Specifies whether this method returns result sets (so that ODBC and JDBC clients can retrieve them).

Usage

To specify that the method returns at least one result set, use the following syntax:

```
ClassMethod name(formal_spec) As returnclass [ ReturnResultsets, SqlName = CustomSets, SqlProc ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that the method returns at least one result set. Set this keyword to true if the method might return one or more result sets. If you do not, then the result sets cannot be retrieved by xDBC clients.

For a stored function, either specify `Not ReturnResultsets` or do not specify this keyword.

Default

If you omit this keyword, result sets cannot be retrieved by xDBC clients.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Stored Procedures" in *Using InterSystems SQL*
- "Stored Functions" in *Using InterSystems SQL*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ServerOnly

Specifies whether this method will be projected to a Java client.

Usage

To override how InterSystems IRIS projects the method to a Java client, use the following syntax:

```
Method name(formal_spec) As returnclass [ ServerOnly=n ]
{    //implementation }
```

Where *n* is one of the following:

- 0 means that this method can be projected.
- 1 means that this method will not be projected.

Details

This keyword specifies that a method will not be projected to a Java client.

Tip

To see which methods of a class are server-only, use the following utility in the Terminal:

```
do dumpMethods^%occLGUtil("Sample.Person")
```

The argument is the fully qualified class name. This utility produces a report that indicates basic information about each method: whether the method is a stub, whether the method is server-only, and (if the method is derived from a property) the property from which it is derived.

Default

If you omit this keyword, this method will not be projected if it is a stub method (but will be projected if it is not a stub method).

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SoapAction

Specifies the SOAP action to use in the HTTP header when invoking this method as a web method via HTTP. Applies only in a class that is defined as a web service or web client.

Usage

To specify the SOAP action to use in the HTTP header when using this method as a web method, use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod, SoapAction = soapaction ]
{    //implementation }
```

Where *soapaction* is one of the following:

- `"[ default ]"` — This causes InterSystems IRIS to use the default value for the SOAP action, which is *NAMESPACE*/*Package.Class.Method*
- `"customValue"` — This causes InterSystems IRIS to use `customValue` as the SOAP action. The value should be a URI that identifies the intent of the SOAP request.

  If you specify a custom value, either it must be unique within for each web method in the web service or you must specify the SoapRequestMessage keyword for each web method (and use unique values for that keyword).
- `""` — This causes InterSystems IRIS to use an empty value as the SOAP action. This scenario is rare.

Details

The SOAP action for a web method is generally used to route the request SOAP message. For example, a firewall could use it to appropriately filter SOAP request messages. An InterSystems IRIS web service service uses the SOAP action, in combination with the message itself, to determine how to process the request message.

This keyword lets you specify the HTTP SOAP action to use when invoking this method as a web method. For SOAP 1.1, the SOAP action is included as the `SOAPAction` HTTP header. For SOAP 1.2, it is included within the `Content-Type` HTTP header.

Default

If you omit the SoapAction keyword, the SOAP action is formed as follows:

```
NAMESPACE/Package.Class.Method
```

Where *NAMESPACE* is the value of the *NAMESPACE* parameter for the web service, *Package.Class* is the name of the web service class, and *Method* is the name of the web method.

Relationship to WSDL

The SoapAction keyword affects the `<binding>` section of the WSDL for the web service. For example, consider the following web method:

```
Method Add(a as %Numeric,b as %Numeric) As %Numeric [ SoapAction = MySoapAction,WebMethod ]
{
    Quit a + b
}
```

For this web service, the `<binding>` section of the WSDL is as follows:

```
<binding name="MyServiceNameSoap" type="s0:MyServiceNameSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="Add">
        <soap:operation soapAction="MySoapAction" style="document"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
```

By default, if the method did not specify the SoapAction keyword, the `<soap:operation>` element might instead be like the following:

```
<soap:operation soapAction="http://www.mynamespace.org/ROBJDemo.BasicWS.Add" style="document"/>
```

If you use the SOAP Wizard to generate an InterSystems IRIS web service service or client from a WSDL, InterSystems IRIS sets this keyword as appropriate for that WSDL.

Effect on the Message

For the web method shown previously, the web service expects a request message of the following form (for SOAP 1.1):

```
POST /csp/gsop/ROBJDemo.BasicWS.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; InterSystems IRIS;)
Host: localhost:8080
Connection: Close
Accept-Encoding: gzip
SOAPAction: MySoapAction
Content-Length: 379
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope >...
```

By default, if the method did not specify the SoapAction keyword, the `SOAPAction` line might instead be like the following:

```
SOAPAction: http://www.mynamespace.org/ROBJDemo.BasicWS.Add
```

Note that for SOAP 1.2, the details are slightly different. In this case, the web service expects a request message of the following form:

```
POST /csp/gsop/ROBJDemo.BasicWS.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; InterSystems IRIS;)
Host: localhost:8080
Connection: Close
Accept-Encoding: gzip
Content-Length: 377
Content-Type: application/soap+xml; charset=UTF-8; action="MySoapAction"

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope >...
```

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapBindingStyle

Specifies the binding style or SOAP invocation mechanism used by this method, when it is used as a web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default binding style used by the method (when it is used as a web method), use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod, SoapBindingStyle = soapbindingstyle ]
{    //implementation }
```

Where *soapbindingstyle* is one of the following:

- `document` (default) — This web method uses document-style invocation.

  With this binding style, the SOAP messages are formatted as documents and typically have only one part.

  In the SOAP messages, the `<Body>` element typically contains a single child element. Each child of the `<Body>` element corresponds to a message part.

- `rpc` — This web method uses RPC (remote procedure call)-style invocation.

  With this binding style, the SOAP messages are formatted as messages with multiple parts.

  In the SOAP messages, the `<Body>` element contains a single child element whose name is taken from the corresponding operation name. This element is a generated wrapper element, and it contains one child element for each argument in the argument list of the method.

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword lets you specify the binding style used by a web method. It affects the format of the SOAP body (but not any SOAP headers).

For a given method, this keyword overrides the SoapBindingStyle class keyword.

Default

If you omit this keyword, the `style` attribute of `<soap:operation>` element is determined instead by the value for the SoapBindingStyle class keyword instead.

Relationship to WSDL

The SoapBindingStyle method keyword specifies the value of the `style` attribute of `<soap:operation>` element within the `<binding>` section of the WSDL. For example, if the SoapBindingStyle method keyword is `document`, the WSDL could look as follows:

```
...
<binding ...>
 ...
    <operation ...>
        <soap:operation ... style="document"/>
...
```

In contrast, if SoapBindingStyle is `rpc`, the WSDL could instead be as follows:

```
...
<binding ...>
 ...
    <operation ...>
        <soap:operation ... style="rpc"/>
...
```

The binding style also affects the request and response <message> elements for the web method, as follows:

- If the binding style is document, each message has only one part by default. For example:

  ```
  <message name="AddSoapIn">
      <part name="parameters" .../>
  </message>
  ```

  If the *ARGUMENTSTYLE* parameter is message, then a message can have multiple parts. For example:

  ```
  <message name="AddSoapIn">
     <part name="a" .../>
     <part name="b" .../>
  </message>
  ```

- If the binding style is rpc, a message can have multiple parts. For example:

  ```
  <message name="AddSoapIn">
      <part name="a" .../>
      <part name="b" .../>
  </message>
  ```

Effect on SOAP Messages

For information, see the entry for the SoapBindingStyle class keyword.

Use with %XML.DataSet

If you use this keyword with a method that uses an object of type %XML.DataSet as input or output, some limitations apply. See the entry for the SoapBindingStyle class keyword.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapBodyUse

Specifies the encoding used by the inputs and outputs of this method, when it is used as a web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default encoding used by the inputs and outputs of the method (when it is used as a web method), use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod, SoapBodyUse = soapbodyuse ]
{    //implementation }
```

Where *soapbodyuse* is one of the following:

- `literal` (default) — This web method uses literal data. That is, the XML within the `<Body>` of the SOAP message exactly matches the schema given in the WSDL.
- `encoded` — This web method uses SOAP-encoded data. That is, the XML within the `<Body>` of the SOAP message uses SOAP encoding as appropriate for the SOAP version being used, as required by the following specifications:
  - SOAP 1.1 (http://www.w3.org/TR/2000/NOTE-SOAP-20000508/)
  - SOAP 1.2 (http://www.w3.org/TR/soap12-part2/)

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword specifies the encoding for the inputs and outputs of a web method.

For a given web method, this keyword overrides the SoapBodyUse class keyword.

Default

If you omit this keyword, the value for the SoapBodyUse class keyword is used instead.

Relationship to WSDL and Effect on SOAP Messages

For information, see the entry for the SoapBodyUse class keyword.

Use with %XML.DataSet

If you use this keyword with a method that uses an object of type %XML.DataSet as input or output, some limitations apply. See the entry for the SoapBindingStyle class keyword.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapMessageName

Specifies the name attribute of the <part> element of the response message for this web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default name of the <part> element of the response message, use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod, SoapMessageName = MyResponse ]
{    //implementation }
```

Where *soapmessagename* is any identifier that is valid in XML.

Details

Note: This keyword has an effect only for a web method that uses SoapBindingStyle equal to `document` (which is the default).

This keyword specifies the name of the child element of the body of the response message.

Default

If you omit this keyword, the message name is the name of the web method with `Response` appended to the end.

The name of the web method is taken from the web method definition in the web service; this can be changed only by renaming that method.

Relationship to WSDL

The SoapMessageName keyword affects the `<messages>` and `<types>` sections of the WSDL for the web service. For example, consider the following web method:

```
Method Add(a as %Numeric,b as %Numeric) As %Numeric [ SoapMessageName=MyResponseMessage,WebMethod ]
{
    Quit a + b
}
```

For this web service, the `<types>` and `<messages>` sections of the WSDL are as follows:

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://www.mynamespace.org">
        <s:element name="Add">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" name="a" type="s:decimal"/>
                    <s:element minOccurs="0" name="b" type="s:decimal"/>
                </s:sequence>
            </s:complexType>
        </s:element>
        <s:element name="MyResponseMessage">
            <s:complexType>
                <s:sequence>
                    <s:element name="AddResult" type="s:decimal"/>
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
</types>
<message name="AddSoapIn">
    <part name="parameters" element="s0:Add"/>
</message>
<message name="AddSoapOut">
    <part name="parameters" element="s0:MyResponseMessage"/>
</message>
```

By default, if the method did not specify the SoapMessageName keyword, the AddSoapOut message would have included an element named `AddResponse` instead of `MyResponseMessage`.

Notice that the SoapMessageName does not affect the child element (for example, `AddResult`) of the response message.

If you use the SOAP Wizard to generate a web service or client from a WSDL, InterSystems IRIS sets this keyword as appropriate for that WSDL.

Effect on SOAP Messages

The web service might send a response message like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Body>
   <MyResponseMessage xmlns="http://www.mynamespace.org">
       <AddResult>42</AddResult>
   </MyResponseMessage>
</SOAP-ENV:Body>
```

By default, if the method did not specify the SoapMessageName keyword, the `<MyResponseMessage>` element would have been `<AddResponse>` instead.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapNameSpace

Specifies the XML namespace used by a web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default XML namespace used by a method (when the method is used as a web method), use the following syntax:

```
Method name(formal_spec) As returnclass [ SoapNameSpace = "soapnamespace", WebMethod ]
{    //implementation }
```

Where *soapnamespace* is a namespace URI. Note that if the URI includes a colon (`:`), the string must be quoted. That is, you can use the following:

```
Method MyMethod() [ SoapNameSpace = "http://www.mynamespace.org", WebMethod ]
```

Or the following:

```
Method MyMethod() [ SoapNameSpace = othervalue, WebMethod ]
```

But not the following:

```
Method MyMethod() [ SoapNameSpace = http://www.mynamespace.org, WebMethod ]
```

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword specifies the XML namespace used by this web method. For details, see *Creating Web Services and Web Clients*.

Note: This keyword has an effect only if the method uses RPC-style binding. That is, the method (or the class that contains it) must be marked with SoapBindingStyle equal to `rpc`. (If you specify this keyword for a method that uses document-style binding, the WSDL will not be self-consistent.)

Default

If you omit this keyword, the method is in the namespace specified by the *NAMESPACE* parameter of the web service or client class.

Relationship to WSDL

For an InterSystems IRIS web service service, the SoapNameSpace keyword affects the namespace declarations within the `<definitions>` element. The namespace that you specify (for example, `http://www.customtypes.org`) is added here. For example:

```
...
xmlns:ns2="http://www.customtypes.org"
xmlns:s0="http://www.wsns.org"
...
targetNamespace="http://www.wsns.org"
```

The `http://www.customtypes.org` namespace is assigned to the prefix `ns2` in this example.

Notice that the WSDL also declares, as usual, the namespace of the web service (`http://www.wsns.org`). This namespace is assigned to the prefix `s0` in this example and is also used as the target namespace.

Effect on SOAP Messages

A possible SOAP message might look as follows (with line breaks and spaces added for readability):

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'
                   xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
                   xmlns:tns='http://www.customtypes.org' >
   <SOAP-ENV:Body SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
      <tns:AddResponse>
         <AddResult>42</AddResult>
      </tns:AddResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the `<AddResponse>` element is in the `http://www.webservicetypesns.org` namespace.

In contrast, if we did not specify the SoapNameSpace keyword, the message would be as follows instead:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'
                   xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
                   xmlns:tns='http://www.wsns.org' >
   <SOAP-ENV:Body SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
      <tns:AddResponse>
         <AddResult>42</AddResult>
      </tns:AddResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this case, the `<AddResponse>` element is in the namespace `http://www.wsns.org`, the namespace of the web service.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapRequestMessage

Use this when multiple web methods have the same SoapAction. This keyword specifies the name of the top element in the SOAP body of the request message, in the default scenario. Applies only in a class that is defined as a web service or web client.

Usage

To specify the name of the top element in the SOAP body of the request message, use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod, SoapAction = "MyAct",
SoapRequestMessage="MyReqMessage" ]
{    //implementation }
```

Where *soaprequestmessage* is a valid XML identifier.

Details

Note: This keyword has an effect only for wrapped document/literal messages.

This keyword specifies the name of the top element in the SOAP body of the request message, for wrapped document/literal messages. (Wrapped document/literal messages are the default. For information, see " Examples of Message Variations" in *Creating Web Services and Web Clients*.)

Specify this keyword if you use the same value for SoapAction for multiple web methods in the same web service. Otherwise, this keyword is not generally needed.

Relationship to WSDL

The SoapRequestMessage keyword affects the `<message>` sections of the WSDL for the web service. For example, consider the following web method:

```
Method Add(a as %Numeric,b as %Numeric) As %Numeric [ SoapAction = MyAct,SoapRequestMessage=MyReqMessage,
 WebMethod ]
{
    Quit a + b
}
```

For this web service, the WSDL includes the following:

```
<message name="AddSoapIn">
   <part name="parameters" element="s0:MyReqMessage"/>
</message>
<message name="AddSoapOut">
   <part name="parameters" element="s0:AddResponse"/>
</message>
```

These elements are correspondingly defined in the `<types>` section.

By default, if the method did not specify the SoapRequestMessage keyword, the `<message>` sections would instead be like the following:

```
<message name="AddSoapIn">
   <part name="parameters" element="s0:Add"/>
</message>
<message name="AddSoapOut">
   <part name="parameters" element="s0:AddResponse"/>
</message>
```

If you use the SOAP Wizard to generate an InterSystems IRIS web service service or client from a WSDL, InterSystems IRIS sets this keyword as appropriate for that WSDL.

Effect on the Message

For the web method shown previously, the web service expects a request message of the following form:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns:s='http://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <MyReqMessage xmlns="http://www.myapp.org"><a>1</a><b>2</b></MyReqMessage>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In contrast, if the method did not specify the SoapRequestMessage keyword, the message would instead be like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xmlns:s='http://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <Add xmlns="http://www.myapp.org"><a>1</a><b>2</b></Add>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapTypeNameSpace

Specifies the XML namespace for the types used by this web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default XML namespace for the types (when the method is used as a web method), use the following syntax:

```
Method name(formal_spec) As returnclass [ SoapTypeNameSpace = "soapnamespace", SoapBindingStyle =
document, WebMethod ]
{    //implementation }
```

Where *soapnamespace* is a namespace URI. Note that if the URI includes a colon (`:`), the string must be quoted. That is, you can use the following:

```
Method MyMethod() [ SoapTypeNameSpace = "http://www.mynamespace.org", SoapBindingStyle = document,
WebMethod ]
```

Or the following:

```
Method MyMethod() [ SoapTypeNameSpace = othervalue, SoapBindingStyle = document, WebMethod ]
```

But not the following:

```
Method MyMethod() [ SoapTypeNameSpace = http://www.mynamespace.org, SoapBindingStyle = document,
WebMethod ]
```

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or a service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword specifies the XML namespace for the types used by this web method. For details, see *Creating Web Services and Web Clients*.

Note: This keyword has an effect only if the method uses document-style binding. That is, the method (or the class that contains it) must be marked with SoapBindingStyle equal to `document`. (It is meaningless to specify this keyword for a method that uses rpc-style binding.)

Default

If you omit this keyword, the types for this method are in the namespace specified by the *TYPENAMESPACE* parameter of the web service or client class. If *TYPENAMESPACE* is not specified, the types are instead in the namespace specified by the *NAMESPACE* parameter of the web service or client.

Relationship to WSDL

The SoapTypeNameSpace keyword affects the following parts of the WSDL:

- The namespace declarations within the `<definitions>` element. The namespace that you specify (for example, `http://www.customtypes.org`) is added here. For example:

```
...
xmlns:ns2="http://www.customtypes.org"
xmlns:s0="http://www.wbns.org"
xmlns:s1="http://webservicetypesns.org"
...
targetNamespace="http://www.wbns.org"
```

The `http://www.customtypes.org` namespace is assigned to the prefix `ns2` in this example.

Notice that the WSDL also declares the following namespaces as usual:

– The namespace of the Web service (`http://www.wsns.org`), which is assigned to the prefix `s0` in this example and which is also used as the target namespace for the Web service.

– The types namespace of the Web service (`http://www.webservicetypesns.org`), which is assigned to the prefix `s1` in this example.

  If no types namespace is specified in the web service class, this namespace is not included in the WSDL.

- The `<types>` element, which includes a `<schema>` element whose `targetNamespace` attribute equals the namespace you specified for SoapTypeNameSpace:

```
<types>
...
<s:schema elementFormDefault="qualified" targetNamespace="http://www.customtypes.org">
    <s:element name="Add">
        <s:complexType>
            <s:sequence>
                <s:element minOccurs="0" name="a" type="s:decimal"/>
                <s:element minOccurs="0" name="b" type="s:decimal"/>
            </s:sequence>
        </s:complexType>
    </s:element>
    <s:element name="AddResponse">
        <s:complexType>
            <s:sequence>
                <s:element name="AddResult" type="s:decimal"/>
            </s:sequence>
        </s:complexType>
    </s:element>
</s:schema>

...
</types>
```

  In contrast, if you did not specify SoapTypeNameSpace, this part of the WSDL would be as follows instead. Notice that the `targetNamespace` for the `<schema>` element is the namespace of the types for the web service:

```
<types>
...
<s:schema elementFormDefault="qualified" targetNamespace="http://www.webservicetypesns.org">
    <s:element name="Add">
        <s:complexType>
            <s:sequence>
                <s:element minOccurs="0" name="a" type="s:decimal"/>
                <s:element minOccurs="0" name="b" type="s:decimal"/>
            </s:sequence>
        </s:complexType>
    </s:element>
    <s:element name="AddResponse">
        <s:complexType>
            <s:sequence>
                <s:element name="AddResult" type="s:decimal"/>
            </s:sequence>
        </s:complexType>
    </s:element>
</s:schema>

...
</types>
```

  (Also, if no types namespace is specified in the web service class, the `targetNamespace` would instead be the namespace of the web service.)

Effect on Messages

A possible SOAP message might look as follows (with line breaks and spaces added for readability):

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'>
   <SOAP-ENV:Body>
      <AddResponse xmlns="http://www.customtypes.org">
         <AddResult>3</AddResult>
      </AddResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the `<AddResponse>` element is in the `"http://www.customtypes.org"` namespace.

In contrast, if we did not specify the SoapTypeNameSpace keyword, the message could be as follows instead:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='http://www.w3.org/2001/XMLSchema'>
   <SOAP-ENV:Body>
      <AddResponse xmlns="http://www.webservicetypesns.org">
         <AddResult>3</AddResult>
      </AddResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SqlName

Overrides the default name of the projected SQL stored procedure. Applies only if this method is projected as an SQL stored procedure.

Usage

To override the default name used when the method is projected as an SQL stored procedure, use the following syntax:

```
ClassMethod name(formal_spec) As returnclass [ SqlProc, SqlName = sqlname ]
{    //implementation }
```

Where *sqlname* is an SQL identifier.

Details

If this method is projected as an SQL stored procedure, then this name is used as the name of the stored procedure.

Default

If you omit this keyword, InterSystems IRIS determines the SQL name as follows:

```
CLASSNAME_METHODNAME
```

This default uses uppercase letters. You can use any case when you invoke the stored procedure, however, because SQL is case-insensitive.

Thus, in the following example, the default SQL name value is `TEST1_PROC1`. This default value is specified in the **SELECT** statement:

```
Class User.Test1 Extends %Persistent
{
ClassMethod Proc1(BO,SUM) As %INTEGER [ SqlProc ]
{
   ///definition not shown
}

Query Q1(KD As %String,P1 As %String,P2 As %String) As %SqlQuery
{
  SELECT SUM(SQLUser.TEST1_PROC1(1,2)) AS Sumd
  FROM SQLUser.Test1
}
}
```

See Also

- "Method Definitions" in this book
- SqlProc keyword
- "Defining Stored Procedures" in *Using InterSystems SQL*
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlProc

Specifies whether the method can be invoked as an SQL stored procedure. Only class methods (not instance methods) can be called as SQL stored procedures.

Usage

To specify that the method can be invoked as an SQL stored procedure, use the following syntax:

```
ClassMethod name(formal_spec) As returnclass [ SqlProc ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that the method can be invoked as an SQL stored procedure. Only class methods (not instance methods) can be called as SQL stored procedures.

Stored procedures are inherited by subclasses.

Default

If you omit this keyword, the method is not available as an SQL stored procedure.

See Also

- "Method Definitions" in this book
- "Defining Stored Procedures" in *Using InterSystems SQL*
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

WebMethod

Specifies whether this method is a web method. Applies only in a class that is defined as a web service or web client.

Usage

To specify that this method is a web method, use the following syntax:

```
Method name(formal_spec) As returnclass [ WebMethod ]
{    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that this method is available as a web method and can be invoked via the SOAP protocol.

Important: In most cases, web methods should be instance methods, rather than class methods. For details and for other requirements for a web method, see *Creating Web Services and Web Clients*.

Default

If you omit this keyword, the method is not available as a web method.

Generated Class

When you add this keyword to a method and compile the class, the class compiler generates an additional class: *Package.OriginalClass.MethodName*, where *Package.OriginalClass* is the class that contains the web method, and *MethodName* is the name of web method.

For example, suppose that you start with the class `ROBJDemo.DocLiteralWS` and you add a method to it named `Add`. When you add the WebMethod keyword to that method and compile, the class compiler generates the class `ROBJDemo.DocLiteralWS.Add`.

Do not modify or directly use this generated class; it is intended only for internal use.

Relationship to WSDL

For a web service, this keyword also affects the generated WSDL, which now contains the additional elements needed to represent this web method.

See Also

- "Method Definitions" in this book
- "Defining and Calling Methods" in *Defining and Using Classes*
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

Parameter Keywords

This reference describes the keywords that apply to a class parameter. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on parameter definitions, see “Parameter Definitions.”

Abstract

Specifies whether this is an abstract parameter.

Usage

To specify that this parameter is abstract, use the following syntax:

```
Parameter name As parameter_type [ Abstract ] = value ;
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

An abstract parameter simply behaves as if it were not defined. However, users can define an abstract parameter for documentation purposes and in order to enforce the signature of this parameter to be defined in the subclasses.

Default

If you omit this keyword, the parameter is not abstract.

See Also

- "Parameter Definitions" in this book
- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Constraint

Specifies a user interface constraint for this parameter.

Usage

To specify a user interface constraint for this parameter, use the following syntax:

```
Parameter name As parameter_type [ Constraint = "constraint" ]  = value ;
```

Where *constraint* is a string used by Studio.

Details

The constraint value is used by Studio to provide input validation for the parameter. Its value is not used nor enforced by the class compiler.

This keyword works in conjunction with the Flags keyword. For example, if Flags is set to ENUM, then Constraint should be a comma-separated list of possible parameter values.

Example

```
Parameter MYPARM [ Constraint = "X,Y,Z", Flags = ENUM ] = X;
```

See Also

- "Parameter Definitions" in this book
- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Deprecated

Specifies that this parameter is deprecated. This keyword is ignored by the class compiler and by Studio, but is used by Atelier.

Usage

To specify that this parameter is deprecated, use the following syntax:

```
Parameter name As parameter_type  [ Deprecated ] = value;
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

See Also

- "Parameter Definitions" in this book

Final

Specifies whether this parameter is final (cannot be overridden in subclasses)

Usage

To specify that a parameter is final, use the following syntax:

```
Parameter name As parameter_type  [ Final ] = value;
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A class member that is marked as *final* cannot be overridden in subclasses.

Default

If you omit this keyword, the parameter is not final.

See Also

- "Parameter Definitions" in this book
- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Flags

Modifies the user interface type (in Studio) for this parameter.

Usage

To modify the user interface type (in Studio) for this parameter, use the following syntax:

```
Parameter name As parameter_type  [ Flags = flags ] = value;
```

Where *flags* is one of the following:

- `ENUM` — The parameter is one of the values specified by the Constraint keyword (a comma-separated list). When you subclass the class that includes this parameter, the Inspector will provide a drop-down list of these values.
- `LIST` — The parameter value is a string consisting of a comma-separated list of items.

Note that `EDIT`, `EMPTY`, and `SYS` are not used.

Details

Modifies the user interface type (in Studio) for the parameter. Studio uses this type to provide input validation for the parameter within the Inspector. The class compiler ignores this keyword.

Default

If you omit this keyword, Studio permits only a single value for the parameter (and does not provide a drop-down list of choices).

See Also

- "Parameter Definitions" in this book
- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this parameter definition is internal (not displayed in the class documentation).

Usage

To specify that this parameter is internal, use the following syntax:

```
Parameter name As parameter_type  [ Internal ] = value;
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this parameter is displayed in the class documentation.

See Also

- "Parameter Definitions" in this book
- "Defining and Referring to Class Parameters" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Projection Keywords

This reference describes the keywords that apply to a class projection. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on projection definitions, see "Projection Definitions."

Internal

Specifies whether this projection definition is internal (not displayed in the class documentation). Note that the class documentation does not currently display projections at all.

Usage

To specify that this projection is internal, use the following syntax:

```
Projection projectionname As class [ Internal ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Note that the class documentation does not currently display projections at all.

Default

If you omit this keyword, the projection is not internal.

See Also

- "Projection Definitions" in this book
- "Defining Class Projections" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Property Keywords

This reference describes the keywords that apply to a property, which you can define in object classes. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on property definitions, see "Property Definitions."

Aliases

Specifies additional names for this property for use via object access.

Usage

To specify additional names for the property, use the following syntax:

```
Property name As classname [ Aliases=othernames ];
```

Where *othernames* is a comma-separated list of valid property names, enclosed in curly braces.

Details

If you specify the Aliases keyword, the compiler creates the given alias or aliases, which point to the same underlying data that the original property points to. For example, suppose we redefine the Name property of Sample.Person as follows:

```
Property Name As %String(POPSPEC = "Name()") [ Aliases = {Alternate}, Required ];
```

Then your code can then work with either the Name property or the equivalent Alternate property, as shown in the following Terminal session:

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

SAMPLES>w p.Name
Fripp,Charles Z.
SAMPLES>w p.Alternate
Fripp,Charles Z.
SAMPLES>set p.Alternate="Anderson,Neville J."

SAMPLES>w p.Name
Anderson,Neville J.
```

Any property methods associated with the original property are also defined for each alias property, so in this example **AlternateIsValid()** is callable and returns the same result as **NameIsValid()** method does. Also if you override a property method (for example, writing a custom **NameGet()** method), then that override automatically applies to the alias property or properties.

Note: This keyword has no effect on the SQL projection of the property.

Default

By default, this keyword is null and a property has no aliases.

Example

```
Property PropA As %String [ Aliases={OtherName,OtherName2} ];
```

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Calculated

Specifies that this property has no in-memory storage allocated for it when the object containing it is instantiated.

Usage

To specify that the property has no in-memory storage allocated for it, use the following syntax:

```
Property name As classname [ Calculated ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies that the property has no in-memory storage allocated for it when the object containing it is instantiated.

Use this keyword when you are defining a property that does not need any in-memory storage. There are two ways to specify a value for this property:

- Define a **Get** (and possibly a **Set**) method for the property. For example, for an Age property, you could provide an **AgeGet** method that determines a person's current age based on the current time and the value of their DateOfBirth property. See the chapter "Using and Overriding Property Methods" in *Defining and Using Classes*.
- Define this property as a computed property; this uses the SqlComputed keyword and related keywords. See "Defining a Computed Property" in *Defining and Using Classes*.

Subclasses inherit the Calculated keyword and cannot override it.

Default

The default value for the Calculated keyword is false.

Example

```
Property Age as %Integer [ Calculated ];
```

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Cardinality

Specifies the cardinality of this relationship property. Required for relationship properties. Not used for other properties.

Usage

To specify the cardinality of a relationship property, use the following syntax:

```
Relationship relname As classname [ Cardinality = cardinality; inverse = inverse ];
```

Where *cardinality* is one of the following:

- `one`
- `many`
- `parent`
- `children`

Details

This keyword specifies the cardinality of a relationship property.

The Cardinality keyword is required for relationship properties. It is ignored by non-relationship properties.

For more information on relationships, see "Defining and Using Relationships" in *Defining and Using Classes.*

Default

There is no default. When you define a relationship, you must specify the Cardinality keyword.

Example

```
Relationship Chapters As Chapter [ Cardinality = many; inverse = Book ];
```

See Also

- "Property Definitions" in this book
- Inverse keyword
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ClientName

Specifies an alias used by client projections of this property.

Usage

To override the default name for this property when the class is projected to a client language, use the following syntax:

```
Property name As classname [ ClientName = clientname ];
```

Where *clientname* is the name to use in the client language.

Details

This keyword lets you define an alternate name for a property when it is projected to a client language. This is especially useful if the property name contains characters that are not allowed in the client language.

Default

If you omit this keyword, the property name is used as the client name.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Collection

Deprecated means of specifying the collection type of a collection property. Do not use.

Details

This keyword has been replaced by the “As” syntax, described in “Working with Collections” in *Defining and Using Classes*.

See Also

- “Property Definitions” in this book
- “Working with Collections” in *Defining and Using Classes*
- “Introduction to Compiler Keywords” in *Defining and Using Classes*

Deprecated

Specifies that this property is deprecated. This keyword is ignored by the class compiler and by Studio, but is used by Atelier.

Usage

To specify that this property is deprecated, use the following syntax:

```
Property name As classname [ Deprecated ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

See Also

- "Property Definitions" in this book

Final

Specifies whether this property is final (cannot be overridden in subclasses).

Usage

To specify that a property is final, use the following syntax:

```
Property name As classname [ Final ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A class member that is marked as *final* cannot be overridden in subclasses.

Default

If you omit this keyword, the property is not final.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Identity

Specifies whether this property corresponds to the identity column in the corresponding SQL table. Applies to persistent classes.

Usage

To specify that this property corresponds to the identity column in the corresponding SQL table, use the following syntax:

```
Property name As %Integer [ Identity ];
```

Note: The type of the property can be any integer type, for example, %BigInt, %Integer, %SmallInt, or %TinyInt.

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

For a persistent object, this keyword specifies that a property corresponds to an identity column in the corresponding SQL table (that is, a column that is marked with the SQL IDENTITY keyword). This keyword is useful particularly for tables that are created through DDL statements. When creating a table using DDL, make sure that any IDENTITY field is defined with `MINVAL=1`, if possible, to allow the creation of a bitmap extent index. See "CREATE TABLE" in the *InterSystems SQL Reference*.

Default

If you omit this keyword, this property is not used as the identity column.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

InitialExpression

Specifies an initial value for this property.

Usage

To specify an initial value for this property, use the following syntax:

```
Property name As classname [ InitialExpression = initialexpression ];
```

Where *initialexpression* is a constant or an ObjectScript expression enclosed in curly braces.

Details

This keyword specifies an initial value for the property. This value is assigned by the **%New()** method of the class when a new instance is created. (If a property is transient, then its initial value is determined either by code invoked by **%New()** when the instance is created or by code invoked by **%OpenId()** when the instance is loaded from disk into memory.)

The value of the initial expression must be suitable for the given property type.

The expression can be arbitrarily complex, with the following limitations:

- The initial expression cannot refer to other properties. That is, an expression such as `{..otherpropertyname}` is not valid.
- The initial expression cannot instantiate an object and cannot include an object reference.
- The initial expression cannot invoke an instance method (only class methods).
- The initial expression must be specified in ObjectScript.
- The code executed by the expression should not report errors. InterSystems IRIS does not provide a way to handle errors returned by the expression.
- If the code executed by the expression causes other processing to occur, InterSystems IRIS does not provide a way to handle results of that processing.

Subclasses inherit the value of the InitialExpression keyword and can override it.

Default

The default value for the InitialExpression keyword is null.

Examples

The following shows several examples that use ObjectScript expressions:

```
Property DateTime As %Date [ InitialExpression = {$zdateh("1966-10-28",3)} ];

Property MyString As %String [ InitialExpression = {$char(0)} ];

/// this one is initialized with the value of a parameter
Property MyProp As %String [ InitialExpression = {..#MYPARM} ];

/// this one is initialized by a class method
Property MyProp2 As %Numeric [ InitialExpression = {..Initialize()} ];
```

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*

- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this property definition is internal (not displayed in the class documentation). .

Usage

To specify that this property is internal, use the following syntax:

```
Property propertyname As classname [ Internal ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this property is displayed in the class documentation.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Inverse

Specifies the inverse side of this relationship. Required for relationship properties. Not used for other properties.

Usage

To specify the relationship property in the related class that is the inverse of this relationship property, use the following syntax:

```
Relationship Chapters As Chapter [ Cardinality = cardinality; Inverse = inverse ];
```

Where *inverse* is the name of the property in the related class.

Details

This keyword specifies the name of the inverse side of a relationship — that is, the name of the corresponding relationship property in the related class. The inverse property must exist in the related class and have the correct Cardinality value.

The Inverse keyword is required for relationship properties. It is ignored by non-relationship properties.

For more information, see "Defining and Using Relationships" in *Defining and Using Classes.*

Default

There is no default. When you define a relationship, you must specify the Inverse keyword.

Example

```
Relationship Chapters As Chapter [ Cardinality = many; inverse = Book ];
```

See Also

- "Property Definitions" in this book
- Cardinality keyword
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

MultiDimensional

Specifies that this property has the characteristics of a multidimensional array.

Usage

To specify that this property has the characteristics of a multidimensional array, use the following syntax:

```
Property Data [ Multidimensional ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A multidimensional property is different from other properties as follows:

- InterSystems IRIS does not provide property methods for it (for information on property methods, see *Defining and Using Classes*).
- It is ignored when the object is validated or saved.
- It is not saved to disk, unless your application includes code to save it specifically.

  That is, the property is also automatically Transient.
- It cannot be exposed to Java or other clients.
- It cannot be stored in or exposed through SQL tables.

Multidimensional properties are rare but provide a useful way to temporarily contain information about the state of an object.

Default

If this keyword is omitted, the property is not multidimensional.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

OnDelete

Specifies the action to take in the current table when a related object is deleted. This keyword applies only to a relationship property that specifies Cardinality as `Parent` or `One`. Its use is invalid in all other contexts.

Usage

To specify the action to take in the current table when a related object is deleted, use the following syntax:

```
Relationship relname As classname [ Cardinality = cardinality, Inverse = inverse, OnDelete = ondelete
 ];
```

Where *ondelete* is one of the following values. In this discussion, *related record* is a record or object belonging to the other side of the relationship, and *referencing record* is the record or object in this side of the relationship.

- `cascade` — When a related record is deleted, the referencing record in this table is also deleted.
- `noaction` — When an attempt is made to delete a related record, the attempt fails.
- `setdefault` — When a related record is deleted, the referencing record in this table is set to its default value.
- `setnull` — When a related record is deleted, the referencing record in this table is set to null.

Details

This keyword defines the referential action that occurs when a record is deleted on the other side of a relationship.

Default

If you omit this keyword, then:

- For a relationship with Cardinality as `Parent`, OnDelete is `cascade`. That is, when you delete the parent record, by default, the associated child records are deleted.
- For a relationship with Cardinality as `One`, OnDelete is `noaction`. That is, when you attempt to delete the "one" record, by default, the attempt fails if the other table has any records that point to it.

Example

```
Relationship Patient As MyApp.Patient [ Cardinality = parent, Inverse = Diagnoses, OnDelete = cascade
 ];
```

See Also

- "Property Definitions" in this book
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Private

Specifies whether the property is private (can be used only by methods of this class or its subclasses).

Usage

To specify that the property is private, use the following syntax:

```
Property name As classname [ Private ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A private class member can only be used by methods of this class (or its subclasses).

A private property is not displayed in the catalog information (accessed by using %Library.SQLCatalog) and is not returned by a SELECT * query. However, you can explicitly refer to and use a private property in an SQL query.

Subclasses inherit the value of the Private keyword and cannot override it.

In InterSystems IRIS, private properties are always inherited and visible to subclasses of the class that defines the property; other languages often call these protected properties.

Default

If you omit this keyword, this property is not private.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ReadOnly

Specifies that a property is read-only, which limits the number of ways its value can be set.

Usage

To specify that the property is read-only, use the following syntax:

```
Property name As classname [ ReadOnly ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Important: Do not use the ReadOnly keyword with collection properties.

Details

This keyword specifies that you cannot set the value of that property by using an object reference. If you attempt to set the value of a read-only property using an object reference, such as:

```
set oref.Name = "newvalue"
```

then there is a <CANNOT SET THIS PROPERTY> error at runtime.

Similarly, when a property is defined as read-only, the field in the corresponding SQL table is also defined as read-only. You cannot explicitly insert or update a read-only field via SQL statements. An attempt to do so results in an SQL error with an SQLCODE of -138.

You can specify the value of a read-only property in the following ways:

- Via the InitialExpression keyword.
- Via the SQLComputeCode keyword.
- Within a property method as described in "Using and Overriding Property Methods" in *Defining and Using Classes*.

Note that each of these techniques has specific limitations.

Notes

If a property is marked as both read-only and required, note the following difference in behavior between object access and SQL access:

- When you save the object, InterSystems IRIS does not validate the property. This means that InterSystems IRIS ignores the Required keyword for that property.
- When you insert or update a record, InterSystems IRIS does consider the Required keyword for the property.

Default

If you omit this keyword, the property is not read-only.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*

- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Required

For a persistent class, specifies that the property's value must be given a value before it can be stored to disk. For an XML-enabled class, specifies that the element to which the property is mapped is required.

Usage

To specify that the property is required, use the following syntax:

```
Property name As classname [ Required ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

For a persistent class, this keyword specifies that the property must be given a value before the containing object can be stored to disk; an error occurs if the property does not have a value. If a property is of type %Stream, the stream cannot be a null stream. That is, the property is considered to have a value if the **%IsNull()** method returns 0.

For a class that extends %XML.Adaptor, this keyword affects the corresponding XML schema. If a property is marked Required, then the corresponding element in the schema does not have `minOccurs="0"` and is thus considered required. See *Projecting Objects to XML*. Note that in this case, the class does not have to be a persistent class. XML schema validation occurs when InterSystems IRIS reads an XML document; see *Using XML Tools*.

In a subclass, you can mark an optional property as required, but you cannot do the reverse.

Notes

If a property is marked as both read-only and required, note the following difference in behavior between object access and SQL access:

- When you save the object, InterSystems IRIS does not validate the property. This means that InterSystems IRIS ignores the Required keyword for that property.
- When you insert or update a record, InterSystems IRIS does consider the Required keyword for the property.

Default

If you omit this keyword, the property is not required.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

ServerOnly

Specifies whether this property is projected to a Java client.

Usage

To specify whether the property is projected to a Java client, use the following syntax:

```
Property name As classname [ ServerOnly = n ];
```

Where *n* is one of the following:

- 0 means that this property is projected.
- 1 means that this property is not projected.

Details

This keyword specifies whether a property is projected to a Java client.

Default

If you omit this keyword, the property is projected.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlColumnNumber

Specifies the SQL column number for this property. Applies only to persistent classes.

Usage

To specify the SQL column number for the property, use the following syntax:

```
Property name As classname [ SqlColumnNumber = 4 ];
```

Where *n* is a positive integer.

Details

This keyword lets you explicitly set the SQL column number for this property. This is provided to support legacy applications.

Default

The default is an empty string.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlComputeCode

Specifies code that sets the value of this property.

Usage

To specify how the property is computed, use the following syntax:

```
Property name As classname [ SqlComputeCode = { Set {FieldName} = Expression }, SqlComputed ];
```

Where:

- *FieldName* — The SQL field name of the property being defined.
- *Expression* — ObjectScript expression that specifies the value of the property.

Details

If this keyword is specified (and if SqlComputed is true), then this property is a computed property. See "Defining a Computed Property" in *Defining and Using Classes*.

For the value of this keyword, specify (in curly braces) a line of ObjectScript code that sets the value of the property, according to the following rules:

- To refer to *this* property, use `{*}`

  Or if the SqlFieldName keyword is *not* specified for the property, use `{propertyname}` where *propertyname* is the property name. If the SqlFieldName keyword *is* specified for the property, use `{sqlfieldnamevalue}` where *sqlfieldnamevalue* is the value of that keyword.

  Note that SqlFieldName is available for all object classes, although it is useful only for persistent classes.

  For further information on field names in ObjectScript code, see "Controlling the SQL Projection of Literal Properties" in the chapter "Defining and Using Literal Properties" in *Defining and Using Classes* or see CREATE TRIGGER in the *InterSystems SQL Reference*.

- Similarly, to refer to another property, if the SqlFieldName keyword is *not* specified for the property, use `{propertyname}` where *propertyname* is the property name. If the SqlFieldName keyword *is* specified for the property, use `{sqlfieldnamevalue}` where *sqlfieldnamevalue* is the value of that keyword.
- The code can include multiple **Set** commands, if necessary. Blank spaces are permitted before or after the equal sign, though each entire **Set** statement must appear on a single line.
- The code can refer to class methods, routines, or subroutines via the usual full syntax. Similarly, it can use ObjectScript functions and operators.
- The code can include embedded SQL.
- The code can include the following pseudo-field reference variables, which are translated into specific values at class compilation time:
  - {%%CLASSNAME} and {%%CLASSNAMEQ} both translate to the name of the class which projected the SQL table definition. {%%CLASSNAME} returns an unquoted string and {%%CLASSNAMEQ} returns a quoted string.
  - {%%TABLENAME} translates to the fully qualified name of the table, returned as a quoted string.
  - {%%ID} translates to the RowID name. This reference is useful when you do not know the name of the RowID field.

  These names are not case-sensitive.

- The code *cannot* use syntax of the form `..`*propertyname* or `..`*methodname()*

For example:

```
Property TestProp As %String [ SqlComputeCode = {set {*} = {OtherField}}, SqlComputed ];
```

For another example:

```
Property FullName As %String [ SqlComputeCode = {set {*}={FirstName}_" "_{LastName}}, SqlComputed ];
```

The code is called with a **Do** command.

Important:

- If you intend to index this field, use deterministic code, rather than nondeterministic code. InterSystems IRIS cannot maintain an index on the results of nondeterministic code because it is not possible to reliably remove stale index key values. (Deterministic code returns the same value every time when passed the same arguments. So for example, code that returns $h is nondeterministic, because $h is modified outside of the control of the function.)
- Any user variables used in the SqlComputeCode should be **New**'d before they are used. This prevents any conflict with variables of the same name elsewhere in related code.

Default

The default is an empty string.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlComputed

Specifies whether that this is a computed property.

Usage

To specify that this property is computed, use the following syntax:

```
Property name As classname [ SqlComputeCode = sqlcomputecode, SqlComputed ];
```

Where *sqlcomputecode* is described in SqlComputeCode.

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

If this keyword is true (and if the property also specifies SqlComputeCode), then this property is a computed property. For options and other details, see "Defining a Computed Property" in *Defining and Using Classes*.

If a property has a value for the SqlComputed keyword, InterSystems IRIS uses that value to compute the property. Specifically, a new class method, **<property>Compute**, is generated from SqlComputeCode. This method is called from the property's **<property>Get** method. If the property also has SqlComputeOnChange keyword specified, then the **<property>Compute** method is called at the specified times.

This functionality is implemented in the **<property>Get** and **<property>Set** methods. If you override either of those methods, then property computations do not work unless there are provisions in the overridden method implementations to trigger computations.

Default

If you omit this keyword, this property is not computed.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlComputeOnChange

This keyword controls when the property is recomputed. Applies only to triggered computed properties.

Usage

To specify when a property is recomputed, use the following syntax:

```
Property name As classname [ SqlComputed, SqlComputeCode=sqlcomputecode, SqlComputeOnChange =
propertynames ];
```

Where *sqlcomputecode* is described in SqlComputeCode and *propertynames* is either a single property name or a comma-separated list of property names. This value can also include the values `%%INSERT` or `%%UPDATE`.

Note that you must use the actual property names, rather than the values given by SqlFieldname.

Details

This keyword applies only to triggered computed properties; it is ignored for other properties. (A triggered computed property is a property for which SqlComputed is true and SqlComputeCode is specified, but for which Calculated and Transient are both false. See "Defining a Computed Property" in *Defining and Using Classes*.)

This keyword controls the conditions under which this property is recomputed. Recomputation can result from:

- The modification of any specified properties.
- The occurrence of a triggering event.

If the keyword has a value of `%%INSERT` or `%%UPDATE`, then **INSERT** or **UPDATE** calls, respectively, specify event-triggered computation of the value of the field (property).

- With `%%INSERT`, InterSystems IRIS computes the field value when a row is inserted into the table. InterSystems IRIS invokes the code specified in the SQLComputeCode keyword to set the value. If SQLComputeCode uses the same field as an input value, then InterSystems IRIS uses the value explicitly provided for that field; if no value is given, InterSystems IRIS uses the InitialExpression (if this is specified) or null (if InitialExpression is not specified).
- With `%%UPDATE`, InterSystems IRIS computes the field value when a row is inserted into the table and recomputes it when a row is updated. In both cases, InterSystems IRIS invokes the code specified in the SQLComputeCode keyword to set the value. If SQLComputeCode uses the same field as an input value, then InterSystems IRIS uses the value explicitly provided for that field; if no value is given, InterSystems IRIS uses the previous field value.

Any event-triggered computation occurs immediately before validation and normalization (which themselves are followed by writing the value to the database).

Note: Event-triggered computation of a field's value may override any explicitly specified value for the property, depending on the code that computes the property's value.

Default

The default value for the SqlComputeOnChange keyword is an empty string.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*

- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlFieldName

Specifies the field name to use in the SQL projection. Applies to persistent classes.

Usage

To override the default name for this property when the table is projected to SQL, use the following syntax:

```
Property name As classname [ SqlFieldName = sqlfieldname ];
```

Where *sqlfieldname* is an SQL identifier.

Details

This keyword specifies the column name used to identify the property in its SQL projection.

Default

If you omit this keyword, the property name is used as the SQL column name.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlListDelimiter

Specifies the delimiter character used within SQL for lists. Applies to list properties in persistent classes. For use only by legacy applications.

Usage

To specify the delimiter character used within SQL for this list property, use the following sytnax:

```
Property Name As List Of Classname [ SqlListDelimiter = """delimiter""", SqlListType = DELIMITED ];
```

Where *delimiter* is a delimiter character.

Details

This keyword specifies the delimiter character used within SQL for this property if it is a list and if SqlListType is `DELIMITED` or `SUBNODE`. This keyword is provided to support legacy applications.

Default

The default value for the SqlListDelimiter keyword is an empty string.

Example

```
Property Things As list Of %String [ SqlListDelimiter = """,""", SqlListType = DELIMITED ];
```

See Also

- "Property Definitions" in this book
- "Working with Collections" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlListType

Specifies the values of this field are represented in memory in SQL and stored on disk. Applies only to list properties in persistent classes. For use only by legacy applications.

Usage

```
Property Name As List Of Classname [ SqlListType = sqllisttype ];
```

Where *sqllisttype* is one of the following:

- `LIST` — the list is stored in $List() format in memory and on disk. This is the default.
- `DELIMITED` — the list is stored as a delimited string in memory and on disk. The delimiter is specified by SqlListDelimiter keyword.
- `SUBNODE` — the list is stored in subnodes on disk; that is, each list element in a separate global node. The in-memory value of the field is $List format if SqlListDelimiter is not specified. If SqlListDelimiter is specified, the in-memory format is a delimited string.

Details

SqlListType controls how the values of a field are represented in memory in SQL, and how they stored on disk.

This keyword is provided to support legacy applications.

Default

The default is `LIST`.

See Also

- "Property Definitions" in this book
- "Working with Collections" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Transient

Specifies whether the property is stored in the database. Applies only to persistent classes.

Usage

To specify that the property is not stored in the database, use the following syntax:

```
Property name As classname [ Transient ];
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

In the case of a persistent class, specifies that the property is not stored in the database.

Note that InterSystems IRIS validates transient properties in the same way as other properties. For example, when you try to save the object, the system validates all its properties, including any transient properties.

In a subclass, you can mark a non-transient property as transient, but you cannot do the reverse.

Default

If this keyword is omitted, the property is not transient.

See Also

See "Defining a Computed Property" in *Defining and Using Classes*.

See Also

- "Property Definitions" in this book
- "Defining and Using Literal Properties" in *Defining and Using Classes*
- "Working with Collections" in *Defining and Using Classes*
- "Working with Streams" in *Defining and Using Classes*
- "Defining and Using Object-Valued Properties" in *Defining and Using Classes*
- "Defining and Using Relationships" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Query Keywords

This reference describes the keywords that apply to a class query. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on query definitions, see "Query Definitions."

ClientName

An alias used by client projections of this query.

Usage

To override the default name for the query when it is projected to a client language, use the following syntax:

```
Query name(formal_spec) As classname [ ClientName = clientname ] {    //implementation }
```

Where *clientname* is the name to use in the client language.

Details

This keyword lets you define an alternate name for a query when it is projected to a client language. This is especially useful if the query name contains characters that are not allowed in the client language.

Default

If you omit this keyword, the query name is used as the client name.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Final

Specifies whether this query is final (cannot be overridden in subclasses).

Usage

To specify that a query is final, use the following syntax:

```
Query name(formal_spec) As classname [ Final ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A class member that is marked as *final* cannot be overridden in subclasses.

Default

If you omit this keyword, the query is not final.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this query definition is internal (not displayed in the class documentation).

Usage

To specify that this query definition is internal, use the following syntax:

```
Query name(formal_spec) As classname [ Internal ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Default

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this query is displayed in the class documentation.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Private

Specifies whether the query is private.

Usage

To specify this query is private, use the following syntax:

```
Query name(formal_spec) As classname [ Private ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Private class members can be used only by other members of the same class (or its subclasses). Note that other languages often use the word *protected* to describe this kind of visibility and use the word *private* to mean invisibility from subclasses.

Default

If you omit this keyword, this query is not private.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SoapBindingStyle

Specifies the binding style or SOAP invocation mechanism used by this query, when it is used as a web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default binding style used by the query (when it is used as a web method), use the following syntax:

```
Query name(formal_spec) As classname [ WebMethod, SoapBindingStyle = soapbindingstyle ] {
//implementation }
```

Where *soapbindingstyle* is one of the following values:

- `document` — This web method uses document-style invocation.

  With this binding style, the SOAP messages are formatted as documents and typically have only one part.

  In the SOAP messages, the `<Body>` element typically contains a single child element. Each child of the `<Body>` element corresponds to a message part.

- `rpc` — This web method uses RPC (Remote Procedure Call)-style invocation.

  With this binding style, the SOAP messages are formatted as messages with multiple parts.

  In the SOAP messages, the `<Body>` element contains a single child element whose name is taken from the corresponding operation name. This element is a generated wrapper element, and it contains one child element for each argument in the argument list of the method.

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword lets you specify the binding style used by this query when it is invoked as a web method.

For a given query, this keyword overrides the SoapBindingStyle class keyword.

Default

If you omit this keyword, the `style` attribute of `<soap:operation>` element is determined instead by the value for the SoapBindingStyle class keyword instead.

Relationship to WSDL

For information, see the entry for the SoapBindingStyle method keyword. (Note that the class keyword of the same name affects more parts of the WSDL than the method keyword and query keyword do.)

Effect on SOAP Messages

For information, see the entry for the SoapBindingStyle class keyword.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

- *Creating Web Services and Web Clients*

SoapBodyUse

Specifies the encoding used by the inputs and outputs of this query, when it is used as a web method. Applies only in a class that is defined as a web service or web client.

Usage

To override the default encoding used by the inputs and outputs of the query (when it is used as a web method), use the following syntax:

```
Query name(formal_spec) As classname [ WebMethod, SoapBodyUse = encoded ] {    //implementation }
```

Where *soapbodyuse* is one of the following values:

- `literal` — This web method uses literal data. That is, the XML within the `<Body>` of the SOAP message exactly matches the schema given in the WSDL.
- `encoded` — This web method uses SOAP-encoded data. That is, the XML within the `<Body>` of the SOAP message uses SOAP encoding as appropriate for the SOAP version being used, as required by the following specifications:
  - SOAP 1.1 (http://www.w3.org/TR/2000/NOTE-SOAP-20000508/)
  - SOAP 1.2 (http://www.w3.org/TR/soap12-part2/)

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword lets you specify the encoding for the inputs and outputs of this query when it is invoked as a web method.

For a given query, this keyword overrides the SoapBodyUse class keyword.

Default

If you omit this keyword, the value for the SoapBodyUse class keyword is used instead.

Relationship to WSDL and Effect on SOAP Messages

For information, see the entry for the SoapBodyUse class keyword.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SoapNameSpace

Specifies the namespace at the binding operation level in the WSDL. Applies only in a class that is defined as a web service or web client.

Usage

To override the default namespace at the binding operation level (when the query is used as a web method), use the following syntax:

```
Query name(formal_spec) As classname [ SoapNameSpace = "soapnamespace", WebMethod ] {    //implementation
 }
```

Where *soapnamespace* is a namespace URI. Note that if the URI includes a colon (`:`), the string must be quoted. That is, you can use the following:

```
Query MyQuery() [ SoapNameSpace = "http://www.mynamespace.org", WebMethod ]
```

Or the following:

```
Query MyQuery() [ SoapNameSpace = othervalue, WebMethod ]
```

But not the following:

```
Query MyQuery() [ SoapNameSpace = http://www.mynamespace.org, WebMethod ]
```

Important: For a web service that you create manually, the default value of this keyword is usually suitable. When you generate a web client or service from a WSDL with the SOAP Wizard, InterSystems IRIS sets this keyword as appropriate for that WSDL; if you modify the value, your web client or service may no longer work.

Details

This keyword lets you specify the XML namespace used by this query when it is invoked as a web method.

Note: This keyword has an effect only if the query uses RPC-style binding. That is, the query (or the class that contains it) must be marked with SoapBindingStyle equal to `rpc`. (If you specify this keyword for a query that uses document-style binding, the WSDL will not be self-consistent.)

For details, see *Creating Web Services and Web Clients*

Default

If you omit this keyword, the web method is in the namespace specified by the *NAMESPACE* parameter of the web service or client class.

Relationship to WSDL and Effect on SOAP Messages

For information, see the entry for the SoapNameSpace method keyword.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

SqlName

Overrides the default name of the projected SQL stored procedure. Applies only if this query is projected as an SQL stored procedure.

Usage

To override the default name used when the query is projected as an SQL stored procedure, use the following syntax:

```
Query name(formal_spec) As classname [ SqlProc, SqlName = sqlname ] {    //implementation }
```

Where *sqlname* is an SQL identifier.

Details

If this query is projected as an SQL stored procedure, then this name is used as the name of the stored procedure.

Default

If you omit this keyword, the query name is used as the SQL procedure name.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlProc

Specifies whether the query can be invoked as an SQL stored procedure.

Usage

To specify that the query can be invoked as an SQL stored procedure, use the following syntax:

```
Query name(formal_spec) As classname [ SqlProc ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies whether the query can be invoked as an SQL stored procedure.

Default

If you omit this keyword, the query cannot be invoked as an SQL stored procedure.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlView

Specifies whether to project this query as an SQL view.

Usage

To specify that the query is projected as an SQL view, use the following syntax:

```
Query name(formal_spec) As classname [SqlView]  {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies whether InterSystems IRIS projects this query as an SQL view.

Default

If this keyword is omitted, InterSystems IRIS does not project this query as an SQL view.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlViewName

Overrides the default name of the projected SQL view. Applies only if this query is projected as an SQL view.

Usage

To override the default name used when the query is projected as an SQL view, use the following syntax:

```
Query name(formal_spec) As classname [ SqlView, SqlViewName = "_Q1" ] {    //implementation }
```

Where *sqlviewname* is an SQL identifier.

Details

This keyword provides an SQL alias for the view projected from this query.

Default

If you omit this keyword, the SQL view name is the query name.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

WebMethod

Specifies whether this class query is a web method. Applies only in a class that is defined as a web service or web client.

Usage

To specify that this query is a web method, use the following syntax:

```
Query name(formal_spec) As classname [ WebMethod ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

This keyword specifies whether this class query is a web method and can be invoked via the SOAP protocol.

For requirements for a web method, see *Creating Web Services and Web Clients*.

Default

If you omit this keyword, the query cannot be invoked as a web method.

Generated Class

When you add this keyword to a class query and compile the class, the class compiler generates two additional classes:

- *Package.OriginalClass.QueryName*
- *Package.OriginalClass.QueryName*`.DS`

Where *Package.OriginalClass* is the class that contains the web method, and *QueryName* is the name of class query.

For example, suppose that you start with the class `ROBJDemo.QueryWS` and you add a class query to it named `MyQuery`. When you add the WebMethod keyword to that class query and compile it, the class compiler generates the following additional classes:

- `ROBJDemo.QueryWS.MyQuery`
- `ROBJDemo.QueryWS.MyQuery.DS`

Do not modify or directly use these generated classes; they are is intended only for internal use.

Relationship to WSDL

For a web service, this keyword also affects the generated WSDL, which now contains the additional elements needed to represent this web method.

See Also

- "Query Definitions" in this book
- "Defining and Using Class Queries" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*
- *Creating Web Services and Web Clients*

Trigger Keywords

This reference describes the keywords that apply to an SQL trigger, which you can define in persistent classes. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on trigger definitions, see "Trigger Definitions."

CodeMode

Specifies how this trigger is implemented.

Usage

To specify how a trigger is implemented, use the following syntax:

```
Trigger name [ Event = sqlevent, CodeMode = codemode ]  {    //implementation }
```

Where *codemode* is one of the following:

- `code` — this trigger is implemented as lines of code (the default).
- `objectgenerator` — this trigger is a trigger generator.

Note: There is an older value for this keyword (`generator`), which is only present for compatibility reasons. Newer applications should use `objectgenerator`.

Details

This keyword specifies how a given trigger is implemented.

By default, the trigger code consists of one or more lines of code to be executed when the trigger is fired.

If CodeMode is `objectgenerator`, however, the trigger is actually a trigger generator. A trigger generator is a program invoked by the class compiler that generates the actual implementation for the given trigger. In this case, the trigger code is responsible for the generated code. The logic is similar to that for method generators; see "Defining Method and Trigger Generators" in *Defining and Using Classes*.

Default

The default value is `code`. That is, by default, a trigger is not a trigger generator.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Event

Specifies the SQL events that will fire this trigger. Required (no default).

Usage

To specify the SQL events that will fire the trigger, use the following syntax:

```
Trigger name [ Event = sqlevent, Time = AFTER ] {    //implementation }
```

Where *sqlevent* is one of the following values:

- `DELETE` — this trigger is fired during an SQL DELETE operation.
- `INSERT` — this trigger is fired during an SQL INSERT operation.
- `UPDATE` — this trigger is fired during an SQL UPDATE operation.
- `INSERT/UPDATE` — this trigger is fired during an SQL Insert operation or an SQL UPDATE operation.
- `INSERT/DELETE` — this trigger is fired during an SQL Insert operation or an SQL DELETE operation.
- `UPDATE/DELETE` — this trigger is fired during an SQL Update operation or an SQL DELETE operation.
- `INSERT/UPDATE/DELETE` — this trigger is fired during an SQL INSERT operation, an SQL UPDATE operation, or an SQL DELETE operation.

Details

This keyword specifies the SQL events that will fire the trigger.

Default

There is no default. When you define a trigger, you must specify a value for this keyword.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Final

Specifies whether this trigger is final (cannot be overridden in subclasses).

Usage

To specify that a trigger is final, use the following syntax:

```
Trigger name [ Event = sqlevent, Final ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

A class member that is marked as *final* cannot be overridden in subclasses.

Default

If you omit this keyword, the trigger is not final.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Foreach

Controls when the trigger is fired.

Usage

To specify when the trigger is fired, use the following syntax:

```
Trigger name [ Event = sqlevent, Foreach = foreach ]  {    //implementation }
```

Where *foreach* is one of the following values:

- `row` — This trigger is fired by each row affected by the triggering statement. Note that row-level triggers are not supported for TSQL, so that the setting of the Language keyword must be `objectscript`.
- `row/object` — This trigger is fired by each row affected by the triggering statement or by changes via object access. Note that row-level triggers are not supported for TSQL, so that the setting of the Language keyword must be `objectscript`.

  This option defines a *unified trigger*, so called because it is fired by data changes that occur via SQL or object access. (In contrast, with other triggers, if you want to use the same logic when changes occur via object access, it is necessary to implement callbacks such as **%OnDelete()**.)
- `statement` — This trigger is fired once for the whole statement. Statement-level triggers are supported for both ObjectScript and TSQL; that is, the setting of the Language keyword can be `objectscript` or `tsql`, respectively.

Details

Controls when the trigger is fired.

Default

If you omit this keyword, the trigger is a row-level trigger.

Exception

Row-level triggers are not supported for TSQL.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Internal

Specifies whether this trigger definition is internal (not displayed in the class documentation).

Usage

To specify that this trigger definition is internal, use the following syntax:

```
Trigger name [ Event = sqlevent, Internal ] {    //implementation }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Default

If you omit this keyword, this trigger is displayed in the class documentation.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Language

Specifies the language in which the trigger is written.

Usage

To specify language in which the trigger is written, use the following syntax:

```
Trigger NewTrigger1 [ Event = sqlevent, Language = language ] {    //implementation }
```

Where *language* is one of the following values:

- `objectscript` — this trigger is written in ObjectScript (default).
- `tsql` — this trigger is written in TSQL. If you use this value, the trigger must be a statement-level trigger; that is, the setting of the Foreach keyword must be `statement`.

Details

This keyword specifies the language in which the trigger is written.

Default

If you omit this keyword, the language is ObjectScript.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

NewTable

Specifies the name of the transition table that stores the new values of the row or statement affected by the event.

Usage

To specify the name of the transition table that stores the new values, use the following syntax:

```
Trigger name [ Event = sqlevent, OldTable = oldtable, NewTable = newtable ] {    //implementation }
```

Where *newtable* is the name of an SQL table in this namespace.

Details

Each trigger has access to the old and new values of the row or statement affected by the event, by means of transition tables (specified by the OldTable and NewTable keywords).

Default

The default value for the NewTable keyword is null.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

OldTable

Specifies the name of the transition table that stores the old values of the row or statement affected by the event.

Usage

To specify the name of the transition table that stores the old values, use the following syntax:

```
Trigger name [ Event = sqlevent, OldTable = oldtable, NewTable = newtable ] {    //implementation }
```

Where *oldtable* is the name of an SQL table in this namespace.

Details

Each trigger has access to the old and new values of the row or statement affected by the event, by means of transition tables (specified by the OldTable and NewTable keywords).

Default

The default is an empty string.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Order

In the case of multiple triggers for the same EVENT and TIME, specifies the order in which the triggers should be fired.

Usage

To specify the order in which this trigger is fired, relative to other triggers with the same EVENT and TIME, use the following syntax:

```
Trigger name [ Event = sqlevent, Order = n, Time = time ] {    //implementation }
```

Where *n* is an integer.

Details

In the case of multiple triggers for the same EVENT and TIME, this keyword specifies the order in which the triggers should be fired.

Default

The default value is 0.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SqlName

Specifies the SQL name to use for this trigger.

Usage

To override the default SQL name of this trigger, use the following syntax:

```
Trigger name [ Event = sqlevent, SqlName = sqlname, Time = time ] {    //implementation }
```

Where *sqlname* is an SQL identifier.

Details

If this trigger is projected to SQL, then this name is used as the SQL trigger.

Default

If you omit this keyword, the SQL trigger name is *triggername*, as specified in the trigger definition.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Time

Specifies whether the trigger is fired before or after the event.

Usage

To specify whether the trigger is fired before or after the event, use the following syntax:

```
Trigger name [ Event = sqlevent, Time = time ] {    //implementation }
```

Where *time* is one of the following:

- `AFTER` — this trigger is fired after an event.
- `BEFORE` — this trigger is fired before an event.

Details

This keyword specifies whether the trigger is fired before or after the event.

Default

The default value is `BEFORE`.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

UpdateColumnList

Specifies one or more columns whose modification causes the trigger to be fired by SQL. Available only for TSQL.

Usage

To specify the columns whose modification fires the trigger, use the following syntax:

```
Trigger name [ Event = sqlevent, UpdateColumnList = updatecolumnlist ]  {    //implementation }
```

Where *updatecolumnlist* is either a column name or comma-separated list of column names, enclosed in parentheses.

Details

This keyword specifies one or more columns whose modification causes the trigger to be fired. Note that this keyword is only available for TSQL.

See Also

- "Trigger Definitions" in this book
- "Defining Method and Trigger Generators" in *Defining and Using Classes*
- "Using Triggers" in *Using InterSystems SQL*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

XData Keywords

This reference describes the keywords that apply to an XData block. These keywords (also known as *class attributes*) generally affect the compiler.

For general information on XData blocks, see "XData Blocks."

Internal

Specifies whether this XData block is internal (not displayed in the class documentation). Note that the class documentation does not currently display XData at all.

Usage

To specify that this XData block is internal, use the following syntax:

```
XData name [ Internal ]  { }
```

Otherwise, omit this keyword or place the word `Not` immediately before the keyword.

Details

Internal class members are not displayed in the class documentation. This keyword is useful if you want users to see a class but not see all its members.

Note that the class documentation does not currently display XData blocks at all.

See Also

- "XData Blocks" in this book
- "Defining and Using XData Blocks" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

MimeType

Specifies the MIME type of the XData block.

Usage

To specify the MIME type of an XData block, use syntax like the following:

```
XData name [ MimeType = mimetype ] { }
```

Where *mimetype* is a valid MIME type (more formally, the Internet media type).

Details

This keyword specifies the MIME type of the contents of the XData block.

Default

The default MIME type is `text/xml`

See Also

- "XData Blocks" in this book
- "Defining and Using XData Blocks" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

SchemaSpec

Specifies the XML schema against which this XData block can be validated.

Usage

To specify an XML schema against which this XData block can be validated, use syntax like the following:

```
XData name [ SchemaSpec = "schemanamespaceURL schemaURL" ] { }
```

Where:

- *schemanamespaceURL* is the URI of the namespace to which the schema belongs
- *schemaURL* is the URL of the schema document

Note that there is a space character between these items. Also note the use of double quotes.

Details

This keyword specifies the XML schema against which this XData block can be validated.

Default

If you omit this keyword, the XData block does not provide an XML schema you can use to validate its contents.

Example

```
XData MyXData [ SchemaSpec = "http:///www.person.com http://www.MyCompany.com/schemas/person.xsd" ]
{
}
```

See Also

- "XData Blocks" in this book
- "Defining and Using XData Blocks" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

XMLNamespace

Specifies the XML namespace to which an XData block belongs.

Usage

To specify the XML namespace to which an XData block belongs, use syntax like the following:

```
XData name [ XMLNamespace = "namespaceURL" ] { }
```

Where *namespaceURL* is the URI of the XML namespace. Note that this item is enclosed in double quotes.

Details

This keyword specifies the XML namespace to which an XData block belongs.

Default

If you omit this keyword, the contents of this XData block do not belong to any namespace.

Example

```
XData MyXData [ XMLNamespace = "http://www.mynamespace.org" ]
{
}
```

See Also

- "XData Blocks" in this book
- "Defining and Using XData Blocks" in *Defining and Using Classes*
- "Introduction to Compiler Keywords" in *Defining and Using Classes*

Storage Keywords

DataLocation

Specifies where data is stored for this class.

```
<DataLocation>^Sample.PersonD</DataLocation>
```

Value

The value of element is a global name with optional leading subscripts.

Description

An expression that is the location where data is stored for this class. Normally this would be a global reference such as `^User.PersonD`. The global reference can also include one or more leading subscripts. For example, `^User.Data("Person")`.

It is also valid to use {%%PARENT} in the place of a global or local variable name in dependent classes (child classes within a parent-child relationship). For example, `{%%PARENT}(ChildData)`. *%%PARENT* evaluates to the parent's ID qualified data location (data location plus parent's ID subscript).

Default Value

The default value for the <DataLocation> element is an empty string in which case the default data location, `^MyApp.MyClassD`, is used (where MyApp.MyClass is the class name).

DefaultData

Specifies the default data storage definition.

```
<DefaultData>MyData</DefaultData>
```

Value

The value of this element is the name of a data storage node within the current storage definition.

Description

Specifies the name of the DATA definition that the class compiler data structure generator uses to place any previously unstored properties. A property is “unstored” if it is storable but is not listed in any DATA definition.

If you add a new, non-transient, property to a persistent class definition, and do not explicitly define a storage location for it, then the class compiler will automatically find a storage location for the property within the storage node specified by the <DefaultData> element.

Default Value

The default value for the <DefaultData> element is an empty string.

Final

Specifies that the storage definition cannot be modified by subclasses.

```
<Final>1</Final>
```

Value

The value of this element is boolean value.

Description

Specifies that the storage definition cannot be modified by subclasses.

Default Value

The default value for the <Final> element is false.

IdLocation

Specifies location of the ID counter.

```
<IdLocation>^Sample.PersonD</IdLocation>
```

Value

The value of this element is a global name with optional leading subscripts.

Description

For non-IDKEY classes, this element lets you specify the global node that contains the counter used to provided object ID values (using the $Increment function).

Default Value

The default value for the <IdLocation> element is an empty string.

IndexLocation

Specifies the default storage location for indices.

```
<IndexLocation>^Sample.PersonI</IndexLocation>
```

Value

The value of this element is a global name with optional leading subscripts.

Description

This element lets you specify the global used for indices for this class. If not specified, the index location is ^*MyApp.MyClassI* (where MyApp.MyClass is the classn ame).

Note that you can also specify the storage of each index individually.

Default Value

The default value for the <IndexLocation> element is an empty string.

SqlRowIdName

Specifies the name used for the row ID within SQL.

```
<SqlRowIdName>IdName</SqlRowIdName>
```

Value

The value of this element is an SQL identifier.

Description

This element lets you directly specify the name of the row (object) ID column projected to SQL.

Default Value

The default value for the <SqlRowIdName> element is an empty string.

SqlRowIdProperty

Specifies the SQL RowId property.

```
<SqlRowIdProperty>prop</SqlRowIdProperty>
```

Value

The value of this element is an SQL identifier.

Description

This element is only used by classes that have been migrated from earlier InterSystems products.

Default Value

The default value for the <SqlRowIdProperty> element is an empty string.

SqlTableNumber

Specifies the internal SQL table number.

```
<SqlTableNumber>123</SqlTableNumber>
```

Value

The value of this element is a table number.

Description

This element is only used by classes that have been migrated from earlier InterSystems products.

Default Value

The default value for the <SqlTableNumber> element is an empty string.

State

Specifies the data definition used for a serial object.

```
<State>state</State>
```

Value

The value of this element is the name of a data definition within this storage definition.

Description

For a serial (embedded) class, this keyword indicates which data definition is used to define the serialized state of the object (how the object's properties are arranged when they are serialized). This is also the default DATA definition to which the default structure generator will add unstored properties.

Default Value

The default value for the <State> element is an empty string.

StreamLocation

Specifies the default storage location for stream properties.

```
<StreamLocation>^Sample.PersonS</StreamLocation>
```

Value

The value of this element is a global name with optional leading subscripts.

Description

This element lets you specify the default global used to store any stream properties within this class. If not specified, the location is *^MyApp.MyClassI* (where MyApp.MyClass is the class name).

Note that you can also specify the storage of each stream property individually; see "Declaring Stream Properties" in the chapter "Working with Streams" in *Defining and Using Classes*.

Default Value

The default value for the <StreamLocation> element is an empty string.

Type

Storage class used to provide persistence.

```
<Type>%Storage.Persistent</Type>
```

Value

The value of this element is a class name.

Description

This element specifies the storage class that provides persistence for this class.

The %Storage.Persistent class is the default storage class and provides the default storage structure.

The %Storage.SQL class is used for mapping classes to legacy data structures.

For serial (embedded) classes, this must be set to %Storage.Serial (which is set automatically by the New Class Wizard).

Default Value

The default value for the <Type> element is %Storage.Persistent.