AS/400 Advanced Series

# IBM Access Class Library for OS/400 Reference

*Version 3*

SC41-4620-00

AS/400 Advanced Series

# IBM Access Class Library for OS/400 Reference

*Version 3*

SC41-4620-00

**Take Note!**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page v.

**First Edition (November 1996)**

This edition applies to the licensed program IBM VisualAge for C++ for AS/400 (Program 5716-CX5), Version 3 Release 7 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions.

Make sure that you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch serving your locality. If you live in the United States, Puerto Rico, or Guam, you can order publications through the IBM Software Manufacturing Solutions at 800+879-2755. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication. You can also mail your comments to the following address:

IBM Corporation
Attention Department 542
IDCLERK
3605 Highway 52 N
Rochester, MN 55901-7829 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

If you have access to Internet, you can send your comments electronically to IDCLERK@RCHVMW2.VNET.IBM.COM; IBMMAIL, to IBMMAIL(USIB56RZ).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

# Contents

## Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the software interoperability coordinator. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Address your questions to:

IBM Corporation
Software Interoperability Coordinator
3605 Highway 52 N
Rochester, MN 55901-7829 USA

This publication could contain technical inaccuracies or typographical errors.

This publication may refer to products that are announced but not currently available in your country. This publication may also refer to products that have not been announced in your country. IBM makes no commitment to make available any unannounced products referred to herein. The final decision to announce any product is based on IBM's business and technical judgment.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This publication contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

## Programming Interface Information

This publication is intended to help you develop applications that use the C++ Access Class Library provided with VisualAge for C++ for AS/400. This publication documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge for C++ for AS/400.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge for C++ for AS/400.

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

| | |
|---|---|
| AS/400 | OS/2 |
| COBOL/400 | OS/400 |
| DB2 | RPG IV |
| DB2/400 | RPG/400 |
| IBM | VisualAge |
| Integrated Language Environment | 400 |

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

# About IBM Access Class Library for OS/400 Reference (SC41-4620)

The classes and class members that are included in the IBM Access Class Library for OS/400 are described in this reference.

IBM Access Class Library for OS/400 consists of classes in the following categories:

- Command
- Data area
- Database
- Data queue
- User space

For information about other AS/400 publications (except Advanced 36), see either of the following:

- The *Publications Reference* book, SC41-4003, in the AS/400 Softcopy Library.
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database that contains descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with the OS/400 operating system at no charge.

For a list of related publications, see the “Bibliography” on page 97.

## Who Should Use This Reference?

This reference is intended for skilled C++ programmers who understand the concept of classes and who are familiar with using C++ templates when working with individual class libraries. Use this reference if you want to access OS/400 resources that are commonly used when constructing client/server applications or programs that must communicate with other programs.

## How to Use This Reference

For detailed information on a particular class or member function, use this reference. If you know what class within a class group a given function is in, you can look at the table of contents entries for that class group, find the class, and look for the member function within the class. If you do not know what class group or class to look in, you can use the index.

Classes are organized alphabetically within each class library. Member functions are listed alphabetically at the start of each class chapter, and their descriptions are grouped according to their purpose. If a class has more than one version of a function, all versions are described in one place.

# Chapter 1. Command Processing Classes

## IC4Command

The IC4Command class is a base class that represents a command. It is the base class for IC4OS400Command that represents a command to be run on an AS/400 system.

### Derivation

IC4Command does not derive from another class.

### Header File

ic4cmd.hpp

## Constructors

IC4Command is an abstract base class. Constructing an instance is not allowed.

## Public Members

### asString

```
virtual IString asString() const =0;
```

Returns the text string that represents a command.

## IC4CommandProcessor

The IC4CommandProcessor class represents an AS/400 system to which commands can be submitted. The command can be an instance of IC4OS400Command or simply a text string.

### Derivation

IC4CommandProcessor does not derive from another class.

### Header File

ic4cp.hpp

## Constructors

```
IC4CommandProcessor(const char* systemName  =NULL);
IC4CommandProcessor(const IC4CommandProcessor&  commandProcessor);
```

The parameters are the following:

*systemName* (in) The name of the AS/400 system that this program is running on. This is an optional parameter. If not specified, the current system name is used.

## IC4CommandProcessor

*commandProcessor* (in) An `IC4CommandProcessor` object. If this object is open, then the newly constructed object is opened.

***Exceptions***

- IC4CommandProcessorException; this applies only to

  **`IC4CommandProcessor`**`(const IC4CommandProcessor& `*`commandProcessor`*`);`

## Public Members

### close

```
virtual void close();
```

Closes the command processor object.

***Exceptions***

- IC4CommandProcessorException

### isOpen

```
IBoolean isOpen() const;
```

Returns True if the command processor is open.

### message

```
virtual IString message();
```

Returns the next message from the last call to run a command. This function can be used to return both error and non-error messages. The string format is `<messageID> - <message text>`. An empty string is returned if there are no messages.

### open

```
virtual void open();
```

Opens the command processor object.

***Exceptions***

- IC4CommandProcessorException

### operator=

```
IC4CommandProcessor& operator=(const IC4CommandProcessor& commandProcessor);
```

Assignment operator: If the CommandProcessor object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4CommandProcessorException

### operator==

```
virtual IBoolean operator==(const IC4CommandProcessor& commandProcessor)  const;
```

Equality operator:  returns True if the system name is the same.

***Exceptions***

- IC4CommandProcessorException

### operator!=

```
virtual IBoolean operator!=(const IC4CommandProcessor& commandProcessor)  const;
```

Inequality operator:  returns True if the system name is not the same.

***Exceptions***

- IC4CommandProcessorException

### operator<

```
virtual IBoolean operator<(const IC4CommandProcessor& commandProcessor)  const;
```

Less-than operator:  returns True if alphabetical order of the system name for the left operand is less than the system name for the right operand.

***Exceptions***

- IC4CommandProcessorException

### operator>

```
virtual IBoolean operator>(const IC4CommandProcessor& commandProcessor)  const;
```

Greater-than operator:  returns True if alphabetical order of the system name for the left operand is greater than the system name for the right operand.

***Exceptions***

- IC4CommandProcessorException

### run

```
virtual void run(const IC4Command& command);
virtual void run(const char* command);
```

Runs a command on an AS/400 system.  The command can be specified by passing an `IC4OS400Command` object or a string containing the command text.

***Exceptions***

- IC4CommandProcessorException

### systemName

```
virtual IString systemName() const;
```

Returns the name of the AS/400 system that was specified on the constructor.

***Exceptions***

- IC4CommandProcessorException

---

## IC4OS400Command

The `IC4OS400Command` class represents a command to be run on an AS/400 system. An instance of `IC4OS400Command` can be passed to an `IC4CommandProcessor` when *run()* is invoked.

### Derivation

```
IC4Command
  IC4OS400Command
```

### Header File

```
ic44cmd.hpp
```

## Constructors

```
IC4OS400Command(const char* commandName);
IC4OS400Command(const IC4OS400Command& command);
```

The parameters are the following:

| | |
|---|---|
| *commandName* | (in) The qualified name of the command, specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYCMD.CMD`. The library can be one of the following:<br>• `a library name`<br>• `%CURLIB%`<br>• `%LIBL%` |
| *command* | (in) An `IC4OS400Command` object. |

***Exceptions***

- IC4CommandException

## Public Members

### addParameter

```
virtual IC4OS400Command& addParameter(const char* keyword,
                                      const char* value);
```

Adds a parameter that is passed when the command is run.

The parameters are the following:

| | |
|---|---|
| *keyword* | (in) Keyword for the parameter. Must be null-terminated. |
| *value* | (in) Value for the parameter. Must be null-terminated. |

***Exceptions***

- IC4CommandException

### asString

```
virtual IString asString() const;
```

Returns the command text as an OS/400 command string.

### name

```
virtual IString name() const;
```

Returns the name of the OS/400 command this object represents as a path name in the library file system.

### numberOfParameters

```
virtual unsigned short numberOfParameters() const;
```

Returns the number of parameters that have been added to this command.

### operator=

```
IC4OS400Command& operator=(const IC4OS400Command& command);
```

Assignment operator.

The parameter is the following:

| | |
|---|---|
| *command* | The IC4OS400Command object from which to do the assignment. |

### operator==

```
virtual IBoolean operator==(const IC4OS400Command& command) const;
```

Equality operator: returns True if the command names are equal.

## IC4OS400Command

### operator!=

```
virtual IBoolean operator!=(const IC4OS400Command& command) const;
```

Inequality operator: returns True if the command names are different.

### operator<

```
virtual IBoolean operator<(const IC4OS400Command& command) const;
```

Less-than operator: returns True if the alphabetical sequence of the command name of the left operand is less than the alphabetical sequence of the command name of the right operand.

### operator>

```
virtual IBoolean operator>(const IC4OS400Command& command) const;
```

Greater-than operator: returns True if the alphabetical sequence of the command name of the left operand is greater than the alphabetical sequence of the command name of the right operand.

### parameter

```
virtual IString parameter(const char* keyword) const;
```

Returns the value of a parameter which was added to the command.

The parameter is the following:

*keyword* (in) The keyword for the parameter. Must be null-terminated.

***Exceptions***

- IC4CommandException

### removeParameter

```
virtual IC4OS400Command& removeParameter(const char* keyword);
```

Removes a parameter.

The parameters are the following:

*keyword* (in) The keyword for the parameter. Must be null-terminated.

***Exceptions***

- IC4CommandException

### removeParameters

```
virtual void removeParameters();
```

Removes all parameters from the command.

## setParameter

```
virtual IC4OS400Command& setParameter(const char* keyword,
                                      const char* value);
```

Sets the value of a parameter to a new value.

The parameters are the following:

| | |
|---|---|
| *keyword* | (in) Keyword for the parameter. Must be null-terminated. |
| *value* | (in) Value for the parameter. Must be null-terminated. |

### *Exceptions*

- IC4CommandException

## IC4OS400Command

# Chapter 2. Data Area Classes

## IC4BaseDataArea

The IC4BaseDataArea class is an abstract base class that represents all AS/400 data areas. Data areas exist in a library on the AS/400 and can have varying attributes. A data area is updated in auxiliary storage whenever it is changed, ensuring that changes are not lost in the event of a program failure. IC4BaseDataArea defines the common member functions for all data areas.

### Derivation

IC4BaseDataArea does not derive from another class.

### Header File

ic4bda.hpp

## Constructors

IC4BaseDataArea is an abstract base class. Constructing an instance is not allowed.

## Public Members

### close

```
virtual void close();
```

Closes the data area object.

***Exceptions***

- IC4DataAreaException

### destroy

```
virtual void destroy();
```

Deletes the data area from the AS/400 system and does an implicit close().

***Exceptions***

- IC4DataAreaException

### isOpen

```
IBoolean isOpen() const;
```

Returns True if the data area is open.

## IC4BaseDataArea

### name
```
virtual const IString& name() const;
```

Returns the fully qualified name of the data area specified on the constructor.

### open
```
virtual void open();
```

Opens the data area object.

***Exceptions***

- IC4DataAreaException

### operator==
```
virtual IBoolean operator==(const IC4BaseDataArea& dataArea) const;
```

Equality operator: returns True if the qualified data area name and system name are equal.

***Exceptions***

- IC4DataAreaException

### operator!=
```
virtual IBoolean operator!=(const IC4BaseDataArea& dataArea) const;
```

Inequality operator: returns True if the qualified data area name and system name are not equal.

***Exceptions***

- IC4DataAreaException

### operator<
```
virtual IBoolean operator<(const IC4BaseDataArea& dataArea) const;
```

Less-than operator:  returns True if the left operand is less than the right operand as determined by the alphabetical order of the system name and then the qualified data area name.

***Exceptions***

- IC4DataAreaException

### operator>
```
virtual IBoolean operator>(const IC4BaseDataArea& dataArea) const;
```

Greater-than operator: returns True if the left operand is greater than the right operand as determined by the alphabetical order of the system name and then the qualified data area name.

***Exceptions***

- IC4DataAreaException

### reset

```
virtual void reset() =0;
```

Sets the contents of the data area to the default creation value for the data area type.

### systemName

```
virtual const IString& systemName() const;
```

Returns the name of the AS/400 system that the program is running on.

---

## IC4CharacterDataArea

The `IC4CharacterDataArea` class represents an AS/400 character data area. Character data areas can contain a character string.

### Derivation

```
IC4BaseDataArea
 IC4CharacterDataArea
```

### Header File

```
ic4cda.hpp
```

## Constructors

```
IC4CharacterDataArea(const char* dataAreaName,
                     const char* systemName =NULL);
```

```
IC4CharacterDataArea(const IC4CharacterDataArea& dataArea);
```

The parameters are the following:

| | |
|---|---|
| *dataAreaName* | (in) The qualified name of the data area specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYDA.DTAARA`. The library can be one of the following:<br>• `a library name`<br>• `%CURLIB%`<br>• `%LIBL%` |
| *systemName* | (in) The name of the AS/400 system that the programming is running on. This is an optional parameter. If not specified, the current system name is used. |

## IC4CharacterDataArea

*dataArea* (in) An `IC4CharacterDataArea` object. If this object is open, then the newly constructed object is opened.

***Exceptions***

- IC4DataAreaException

## Public Members

### create

```
void create(unsigned long length =32,
            char* initialValue =NULL,
            char* textDescription =NULL,
            char* authority =NULL);
```

Creates a character data area on the AS/400. The data area can be created in a specific library or in the current library.

The parameters are the following:

*length* (in) The maximum length the data area can be. This is an optional parameter. The default value is 32 characters. Allowable values are 1-2000.

*initialValue* (in) The initial value given to the data area when it is created. This is an optional parameter. The default value is a blank string.

*textDescription* (in) Description of the data area. This is an optional parameter. The default value is a blank string.

*authority* (in) The public access authority given to users who do not have specific authority to the data area. This is an optional parameter that defaults to `*LIBCRTAUT`. Other possible values are:

- `*CHANGE`
- `*ALL`
- `*USE`
- `*EXCLUDE`
- `authorization list name`

***Exceptions***

- IC4DataAreaException

### dataLength

```
unsigned long dataLength();
```

Returns the maximum length for the data area in bytes.

***Exceptions***

- IC4DataAreaException

## operator=

```
IC4CharacterDataArea& operator=(const IC4CharacterDataArea& dataArea);
```

Assignment operator: If the character data area object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4DataAreaException

## overwrite

```
void overwrite(const char* writeData,
               unsigned short writeDataLength =0);
```

Writes a new value to the data area, overwriting all the data currently in the data area. If the *writeDataLength* is less than the length of the data area, the remainder of the data area is blank padded.

The parameters are the following:

| | |
|---|---|
| *writeData* | (in) The value to write to the data area.<br>This value can contain null characters. |
| *writeDataLength* | (in) The length of the data to write. Possible values are 1-2000. If this value is 0, *writeData* must be null-terminated. |

***Exceptions***

- IC4DataAreaException

## read

```
void read(IString& readData,
          unsigned long readLength =1,
          unsigned long startingPosition =1);
void read(char* readData,
          unsigned long readLength,
          unsigned long startingPosition =1);
```

Reads the data area and returns the amount of data requested.

The parameters are the following:

| | |
|---|---|
| *readData* | (out) String to receive the contents of the data area. |
| *readLength* | (in) The number of characters to read from the data area. Possible values are 1-2000. |

*startingPosition* (in) The 1-based position at which to start reading the data area. This is an optional parameter with a default value of 1.

***Exceptions***

- IC4DataAreaException

### reset

```
virtual void reset();
```

Sets the contents of the data area to blanks.

***Exceptions***

- IC4DataAreaException

### write

```
void write(const char* writeData,
           unsigned long writeDataLength =0,
           unsigned long startingPosition =1);
```

Writes data to the data area, replacing data from the specified starting position for the length of the *writeData* string.

The parameters are the following:

*writeData* (in) The value to write to the data area. This value can contain null characters.

*writeDataLength* (in) The length of the data to write. If this value is 0, *writeData* must be null-terminated.

*startingPosition* (in) The 1-based position in the data area at which to begin writing the new data. The default is to start writing at the first position in the data area.

***Exceptions***

- IC4DataAreaException

## Inherited Public Members

| Member | Class | Page |
|---|---|---|
| close | IC4BaseDataArea | 9 |
| destroy | IC4BaseDataArea | 9 |
| isOpen | IC4BaseDataArea | 9 |
| name | IC4BaseDataArea | 10 |
| open | IC4BaseDataArea | 10 |
| operator== | IC4BaseDataArea | 10 |
| operator!= | IC4BaseDataArea | 10 |
| operator< | IC4BaseDataArea | 10 |
| operator> | IC4BaseDataArea | 10 |

| Member | Class | Page |
|---|---|---|
| systemName | IC4BaseDataArea | 11 |

## IC4DecimalDataArea

The IC4DecimalDataArea class represents an AS/400 decimal data area. A decimal data area contains a decimal value.

### Derivation

IC4BaseDataArea
 IC4DecimalDataArea

### Header File

ic4dda.hpp

## Constructors

```
IC4DecimalDataArea(const char* dataAreaName,
                   const char* systemName =NULL);

IC4DecimalDataArea(const IC4DecimalDataArea& dataArea);
```

The parameters are the following:

| | |
|---|---|
| *dataAreaName* | (in) The qualified name of the data area specified as a path name in the library file system. For example, /QSYS.LIB/MYLIB.LIB/MYDA.DTAARA. The library value can be one of the following:<br>• a library name<br>• %CURLIB%<br>• %LIBL% |
| *systemName* | The name of the AS/400 system that the program is running on. This is an optional parameter. If not specified, the current system name is used. |
| *dataArea* | (in) An IC4DecimalDataArea object. If this object is open, then the newly constructed object is opened. |

***Exceptions***

- IC4DataAreaException

## Public Members

### create

```
void create(unsigned long length =15,
            unsigned long decimalPositions =5,
            double initialValue =0.0,
            char* textDescription =NULL,
            char* authority =NULL);
```

## IC4DecimalDataArea

Creates a decimal data area on an AS/400 system. The data area can be created in a specific library or in the current library.

The parameters are the following:

| | |
|---|---|
| *length* | (in) The maximum length the data area can be. This is an optional parameter. The default value is 15 digits. Allowable values are 1-24. |
| *decimalPositions* | (in) The number of decimal positions in the data area. This is an optional parameter. The default value is 5 decimal positions. Allowable values are 0-9. |
| *initialValue* | (in) The initial value given to the data area when it is created. This is an optional parameter. The default value is 0.0. |
| *textDescription* | (in) Description of the data area. This is an optional parameter. The default value is a blank string. |
| *authority* | (in) The public access authority given to users who do no have specific authority to the data area. This is an optional parameter that defaults to `*LIBCRTAUT`. Other possible values are:<br>• `*CHANGE`<br>• `*ALL`<br>• `*USE`<br>• `*EXCLUDE`<br>• `authorization list name` |

***Exceptions***

- IC4DataAreaException

### dataLength

```
unsigned long dataLength();
```

Returns the maximum length for the data area.

***Exceptions***

- IC4DataAreaException

### decimalPositions

```
unsigned long decimalPositions();
```

Returns the maximum number of decimal positions that a value in the data area may have.

***Exceptions***

- IC4DataAreaException

### operator=

```
IC4DecimalDataArea& operator=(const IC4DecimalDataArea& dataArea);
```

Assignment operator: If the decimal data area object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4DataAreaException

### read

```
void read(double& readData);
```

Reads the contents of a decimal data area.

The parameters are the following:

*readData* (out) The variable that is to contain the data that is read.

***Exceptions***

- IC4DataAreaException

### reset

```
virtual void reset();
```

Sets the contents of a decimal data area to 0.0.

***Exceptions***

- IC4DataAreaException

### write

```
void write(double writeData);
```

Writes a new value to a decimal data area.

The parameters are the following:

*writeData* (in) The value to write to the data area.

***Exceptions***

- IC4DataAreaException

## Inherited Public Members

| Member | Class | Page |
|---|---|---|
| close | IC4BaseDataArea | 9 |
| destroy | IC4BaseDataArea | 9 |
| isOpen | IC4BaseDataArea | 9 |

## IC4LogicalDataArea

| Member | Class | Page |
|---|---|---|
| name | IC4BaseDataArea | 10 |
| open | IC4BaseDataArea | 10 |
| operator== | IC4BaseDataArea | 10 |
| operator!= | IC4BaseDataArea | 10 |
| operator< | IC4BaseDataArea | 10 |
| operator> | IC4BaseDataArea | 10 |
| systemName | IC4BaseDataArea | 11 |

## IC4LogicalDataArea

The `IC4LogicalDataArea` class represents an AS/400 logical data area. The value of a logical data area is always 0 or 1.

### Derivation

```
IC4BaseDataArea
  IC4LogicalDataArea
```

### Header File

`ic4lda.hpp`

## Constructors

```
IC4LogicalDataArea(const char* dataAreaName,
                   const char* systemName =NULL);
```

```
IC4LogicalDataArea(const IC4LogicalDataArea& dataArea);
```

The parameters are the following:

*dataAreaName* (in) The qualified name of the data area specified as a path name in the library file system. For example, `/QSYS.LIB/MYLIB.LIB/MYDA.DTAARA`. The library value can be one of the following:
- `a library name`
- `%CURLIB%`
- `%LIBL%`

*systemName* (in) The name of the AS/400 system that the program is running on. This is an optional parameter. If not specified, the current system name is used.

*dataArea* (in) An `IC4LogicalDataArea` object. If this object is open, then the newly constructed object is opened.

***Exceptions***

- IC4DataAreaException

## Public Members

### create

```
void create(char initialValue ='0',
            char* textDescription =NULL,
            char* authority =NULL);
```

Creates a logical data area on an AS/400 system. The data area can be created in a specific library or in the current library.

The parameters are the following:

| | |
|---|---|
| *initialValue* | (in) The initial value given to the data area when it is created. This is an optional parameter. The default value is 0. |
| *textDescription* | (in) Description of the data area. This is an optional parameter. The default value is a blank string. |
| *authority* | (in) The public access authority given to users who do no have specific authority to the data area. This is an optional parameter that defaults to `*LIBCRTAUT`. Other possible values are:<br>• `*CHANGE`<br>• `*ALL`<br>• `*USE`<br>• `*EXCLUDE`<br>• `authorization list name` |

***Exceptions***

- IC4DataAreaException

### operator=

```
IC4LogicalDataArea& operator=(const IC4LogicalDataArea& dataArea);
```

Assignment operator: If the logical data area object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4DataAreaException

### read

```
void read(char& readData);
```

Reads the contents of a logical data area.

***Exceptions***

- IC4DataAreaException

## IC4LogicalDataArea

### reset

```
virtual void reset();
```

Sets the contents of a logical data area to '0'.

***Exceptions***

- IC4DataAreaException

### write

```
void write(char writeData);
```

Writes data to a logical data area.

The parameter is the following:

*writedata* (in) The value to write to the data area.

***Exceptions***

- IC4DataAreaException

## Inherited Public Members

| Member | Class | Page |
|---|---|---|
| close | IC4BaseDataArea | 9 |
| destroy | IC4BaseDataArea | 9 |
| isOpen | IC4BaseDataArea | 9 |
| name | IC4BaseDataArea | 10 |
| open | IC4BaseDataArea | 10 |
| operator== | IC4BaseDataArea | 10 |
| operator!= | IC4BaseDataArea | 10 |
| operator< | IC4BaseDataArea | 10 |
| operator> | IC4BaseDataArea | 10 |
| systemName | IC4BaseDataArea | 11 |

# Chapter 3. Database Classes

The database access classes are a set of classes that use the DB2 for OS/400 SQL Call Level Interface to access data in the AS/400 database.

Many of the member functions defined in the database access classes use DB2 for OS/400 APIs. Some of these member functions have parameters that accept predefined constant values. For more information on the APIs used in the database access classes, including possible values for parameters, see the *DB2 for OS/400 SQL Call Level Interface,* SC41-4806. For additional information, refer to the *DB2 for OS/400 SQL Programming* book, SC41-4611.

The following tables show all of the member functions supported by the Access Class Library, whether the member function is available on the OS/400 server version, and the DB2 for OS/400 API that is used.

*Table 1. IC4SQLDatabase*

| Member Function | DB2 for OS/400 API | Supported |
|---|---|---|
| columnPrivileges | | No |
| columns | SQLColumns | Yes |
| commit | SQLEndTran | Yes |
| connect | SQLConnect | Yes |
| connectOption | SQLGetConnectOption | Partial |
| disconnect | SQLDisconnect | Yes |
| foreignKeys | | No |
| functionSupported | SQLGetFunctions | Yes |
| getInformation | SQLGetInfo | Yes |
| getStatusInformation | SQLError | Yes |
| nativeSql | None | Yes |
| primaryKeys | | No |
| procedureColumns | | No |
| procedures | | No |
| rollback | SQLEndTran | Yes |
| setConnectOption | SQLSetConnectOption | Partial |
| specialColumns | SQLSpecialColumns | Yes |
| statistics | SQLStatistics | Yes |
| tablePrivileges | | No |
| tables | SQLTables | Yes |
| typeInformation | | No |

## Database Classes

*Table 2. IC4SQLEnvironment*

| Member Function | DB2 for OS/400 API | Supported |
|---|---|---|
| commitDatabases | SQLTransact | Yes |
| datasources | | No |
| getStatusInformation | SQLError | Yes |
| rollbackDatabases | SQLTransact | Yes |

*Table 3. IC4SQLResultSet*

| Member Function | DB2 for OS/400 API | Supported |
|---|---|---|
| bindColumn | SQLBindCol | Partial |
| bindColumns | SQLBindCol | Yes |
| bindToRow | SQLBindCol | Yes |
| close | SQLFreeStmt | Yes |
| columnAttribute | SQLColAttributes | Yes |
| cursorName | SQLGetCursorName | Yes |
| describeColumn | SQLDescribeCol | Yes |
| extendedFetch | | No |
| fetch | SQLFetch | Yes |
| getData | SQLGetData | Partial |
| getStatusInformation | SQLError | Yes |
| moreResults | | No |
| numberResultColumns | SQLNumResultCols | Yes |
| parameterData | SQLParamData | Yes |
| putData | SQLPutData | Yes |
| rowAsString | SQLGetData | Yes |
| rowsAffected | SQLRowCount | Yes |
| setPosition | | No |
| unbind | SQLFreeStmt | Yes |

*Table 4 (Page 1 of 2). IC4SQLStatement*

| Member Function | DB2 for OS/400 API | Supported |
|---|---|---|
| bindParameter | SQLSetParam | Partial |
| bindParameters | SQLSetParam | Yes |
| cancel | SQLCancel | Yes |
| execute | SQLExecute | Yes |
| executeDirect | SQLExecDirect | Yes |
| getParameterDescription | | No |

| Table 4 (Page 2 of 2). IC4SQLStatement | | |
|---|---|---|
| **Member Function** | **DB2 for OS/400 API** | **Supported** |
| getStatusInformation | SQLError | Yes |
| moreResults | | No |
| numberParameters | | No |
| parameterData | SQLParamData | Yes |
| prepare | SQLPrepare | Yes |
| putData | SQLPutData | Yes |
| rowsAffected | SQLRowCount | Yes |
| setCursorName | SQLSetCursorName | Yes |
| setParameterOptions | | No |
| setStatementOption | SQLSetStmtOption | Yes |
| statementOption | SQLGetStmtOption | Yes |
| unbindParameters | SQLFreeStmt | Yes |
| unprepare | SQLFreeStmt | Yes |

## IC4SQLDatabase

The `IC4SQLDatabase` class represents a connection to the database on an AS/400 system. Objects of this class manage a connection to an AS/400 system and provide catalog functions that can be used to query information about the AS/400 database.

### Derivation

`IC4SQLDatabase` does not derive from another class.

### Header File

`ic4sqdb.hpp`

## Constructors

**`IC4SQLDatabase`**`(const IC4SQLEnvironment& `*`environment`*`);`

The parameter is the following:

*environment* The IC4SQLEnvironment object that was instantiated for the application.

***Exceptions***

- IC4SQLException

## IC4SQLDatabase

## Public Members

### columnPrivileges

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### columns

```
virtual RETCODE columns(IC4SQLResultSet& results,
                        const unsigned char* tableQualifier,
                        short tableQualifierLength,
                        const unsigned char* tableOwner,
                        short tableOwnerLength,
                        const unsigned char* tableName,
                        short tableNameLength,
                        const unsigned char* columnName,
                        short columnNameLength);
```

Generates a result set that contains information about the columns in a table that satisfy certain criteria.

The parameters are the following:

| | |
|---|---|
| *results* | (out) The `IC4SQLResultSet` object to receive the results. |
| *tableQualifier* | (in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string. |
| *tableQualifierLength* | (in) Length of table qualifier. For an AS/400 system, this is the library name. |
| *tableOwner* | (in) Name of table owner. For an AS/400 system, this is the library name. |
| *tableOwnerLength* | (in) Length of owner name. |
| *tableName* | (in) Name of table for which information is to be retrieved. |
| *tableNameLength* | (in) Length of table name. |
| *columnName* | (in) String search pattern for column names. |
| *columnNameLength* | (in) Length of string search pattern. |

***API used:*** SQLColumns

#### *Return Values*

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

#### *Exceptions*

- IC4SQLException

### commit

```
virtual RETCODE commit();
```

Commits changes initiated by all statements active under this database object.

***API used:*** SQLEndTran

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## connect

```
virtual RETCODE connect(const char* dataSource,
                        const char* userid =NULL,
                        const char* passwd =NULL);
```

Connects to a database.

The parameters are the following:

| | |
|---|---|
| *dataSource* | (in) The name of the system. |
| *userid* | (in) The userid (authorization name). This name must be the same as the user profile under which the application is running. The value must be null terminated. This is an optional parameter. |
| *passwd* | (in) The password (authentication string). This password must be the same as the user profile under which the application is running. The value must be null terminated. This is an optional parameter. |

***API used:*** SQLConnect

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## connectOption

```
virtual RETCODE connectOption(unsigned short option,
                              unsigned long& value);
```

Returns the current value of a connection option. For information about connection options and the data types returned, refer to the *DB2 for OS/400 SQL Call Level Interface* for API SQLGetConnectOption.

The parameters are the following:

| | |
|---|---|
| *option* | (in) The connection option to retrieve |
| *value* | (out) The value of the connection option |

***API used:*** SQLGetConnectOption

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND

***Exceptions***

- IC4SQLException

## disconnect

```
virtual RETCODE disconnect();
```

Disconnects from the database. If commitment control is on (this is the default) and there are incomplete transactions, call `commit()` or `rollback()` first.

***API used:*** SQLDisconnect

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## foreignKeys

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## functionSupported

```
virtual RETCODE functionSupported(unsigned short function,
                                  unsigned short* supported);
```

This member function returns a value indicating whether the call level interface supports the specified function.

The parameters are the following:

| | |
|---|---|
| *function* | (in) The function being queried. |
| *supported* | (out) User-allocated storage to receive the information. |

***API used:*** SQLGetFunctions

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## getInformation

```
virtual RETCODE getInformation(short infoType,
                               void* infoValue,
                               short infoValueLength,
                               short& bytesAvailable);
```

Gets information, including data conversions, about the DBMS to which the application is currently connected.

The parameters are the following:

| | |
|---|---|
| *infoType* | (in) Type of information desired. |
| *infoValue* | (out) User-allocated storage to receive the information. |
| *infoValueLength* | (in) Length of *infoValue*. |
| *bytesAvailable* | (out) Actual length of information value available to return. |

***API used:*** SQLGetInfo

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned SQL_SUCCESS_WITH_INFO. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns SQL_NO_DATA_FOUND. If the

last operation returned SQL_SUCCESS or threw an exception, SQL_NO_DATA_FOUND is returned.

The parameters are the following:

| | |
|---|---|
| *sqlState* | (out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long. |
| *nativeError* | (out) User-allocated storage to receive the native error value. |
| *messageText* | (out) User-allocated storage to receive the error text. |
| *messageTextLength* | (in) Length of messageText. |
| *bytesAvailable* | (out) Length of text available to return. |

***API used:*** SQLError

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND

***Exceptions***

- IC4SQLException

## nativeSql

```
virtual RETCODE nativeSql(const unsigned char* inputString,
                          long inputStringLength,
                          IString& resultString);
virtual RETCODE nativeSql(const unsigned char* inputString,
                          long inputStringLength,
                          unsigned char* resultString,
                          long resultStringLength,
                          long& resultStringBytesAvailable);
```

This method returns the SQL string as it was supplied to the member function.

The parameters are the following:

| | |
|---|---|
| *inputString* | (in) SQL text string to translate. |
| *inputStringLength* | (in) Length of text string to translate. |
| *resultString* | (out) User-allocated storage to receive the translated string. |
| *resultStringLength* | (in) Length of *resultString*. |
| *resultStringBytesAvailable* | (out) Length of translated string available to return. |

***API used:*** None

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## primaryKeys

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## procedureColumns

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## procedures

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## rollback

```
virtual RETCODE rollback();
```

Rolls back changes initiated by all statements active under this database object.

***API used:*** SQLEndTran

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

## setConnectOption

```
virtual RETCODE setConnectOption(const unsigned short option,
                                 const signed long value);
```

Sets a connection option. Two versions are provided: one for specifying a long option and one for providing a char * option value. If the option specified does not correspond to the type of the value, an exception is thrown.

The parameters are the following:

| | |
|---|---|
| *option* | (in) The connection option to set. |
| *value* | (in) The value of the connection option. |

***API used:*** SQLSetConnectOption

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## specialColumns

```
virtual RETCODE specialColumns(IC4SQLResultSet& results,
                               unsigned short columnType,
                               const unsigned char* tableQualifier,
                               short tableQualifierLength,
                               const unsigned char* tableOwner,
                               short tableOwnerLength,
                               const unsigned char* tableName,
                               short tableNameLength,
                               unsigned short scope,
                               unsigned short nullable);
```

Generates a result set with information about columns in the specified table that either uniquely identify a row in the table or are automatically updated when any value in the row is changed.

The parameters are the following:

| | |
|---|---|
| *results* | (out) The IC4SQLResultSet object to receive the results. |
| *columnType* | (in) Type of information to return. Allowable values are:<br>• SQL_BEST_ROWID<br>• SQL_ROWVER |
| *tableQualifier* | (in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string. |
| *tableQualifierLength* | (in) Length of table qualifier. |
| *tableOwner* | (in) Name of table owner. For an AS/400 system, this is the library name. |
| *tableOwnerLength* | (in) Length of owner name. |
| *tableName* | (in) Name of table for which information is to be retrieved. |
| *tableNameLength* | (in) Length of table name. |
| *scope* | (in) Minimum required scope. Allowable values are:<br>• SQL_SCOPE_CURROW<br>• SQL_SCOPE_TRANSACTION |
| *nullable* | (in) Whether to return nullable columns. Allowable values are:<br>• SQL_NULLABLE<br>• SQL_NO_NULLS |

***API used:*** SQLSpecialColumns

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## statistics

```
virtual RETCODE statistics(IC4SQLResultSet& results,
                           const unsigned char* tableQualifier,
                           short tableQualifierLength,
                           const unsigned char* tableOwner,
                           short tableOwnerLength,
                           const unsigned char* tableName,
                           short tableNameLength,
                           unsigned short unique,
                           unsigned short accuracy);
```

Generates a result set with statistics about a table and the indexes associated with the table.

The parameters are the following:

| | |
|---|---|
| *results* | (out) The IC4SQLResultSet object to receive the results. |
| *tableQualifier* | (in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string. |
| *tableQualifierLength* | (in) Length of table qualifier. |
| *tableOwner* | (in) Name of table owner. For an AS/400 system, this is the library name. |
| *tableOwnerLength* | (in) Length of owner name. |
| *tableName* | (in) Name of table for which information is to be retrieved. |
| *tableNameLength* | (in) Length of table name. |
| *unique* | (in) Type of index. Allowable values are:<br>• SQL_INDEX_UNIQUE<br>• SQL_INDEX_ALL |
| *accuracy* | (in) Importance of CARDINALITY and PAGES. Allowable values are:<br>• SQL_ENSURE<br>• SQL_QUICK |

***API used:*** SQLStatistics

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## tablePrivileges

This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## tables

```
virtual RETCODE tables(IC4SQLResultSet& results,
                       const unsigned char* tableQualifier,
                       short tableQualifierLength,
                       const unsigned char* tableOwner,
                       short tableOwnerLength,
                       const unsigned char* tableName,
                       short tableNameLength,
                       const unsigned char* tableType,
                       short tableTypeLength);
```

Generates a result set with a list of the tables in the database that match the value specified for *tableType*.

The parameters are the following:

| | |
|---|---|
| *results* | (out) The `IC4SQLResultSet` object to receive the results. |
| *tableQualifier* | (in) The table qualifier. For an AS/400 system, this is the RDBNAME that was set using the ADDRDBDIRE CL command or a null string. |
| *tableQualifierLength* | (in) Length of table qualifier. |
| *tableOwner* | (in) Name of table owner. For an AS/400 system, this is the library name. |
| *tableOwnerLength* | (in) Length of owner name. |
| *tableName* | (in) Name of table for which information is to be retrieved. |
| *tableNameLength* | (in) Length of table name. |
| *tableType* | (in) List of table types to match. |
| *tableTypeLength* | (in) Length of tableType. |

***API used:*** SQLTables

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

### typeInformation
This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

---

## IC4SQLEnvironment

The `IC4SQLEnvironment` class represents the environment that provides global functions such as committing or rolling back changes for all database objects created under the environment. An IC4SQLEnvironment object is required as input to the constructor for an IC4SQLDataBase object.

### Derivation
`IC4SQLEnvironment` does not derive from another class.

### Header File
`ic4sqenv.hpp`

## Constructors

**`IC4SQLEnvironment();`**

Only one environment can be constructed by an application.

***Exceptions***

- IC4SQLException

## Public Members

### commitDatabases
`virtual RETCODE` **`commitDatabases();`**

Commits all transactions active across all connected `IC4SQLDatabase` objects.

***API used:*** SQLTransact

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

### dataSources
This function is not supported. An `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## IC4SQLEnvironment

### environmentOption

```
RETCODE environmentOption(long option,
                          signed long& value);
RETCODE environmentOption(long option,
                          char* value,
                          long valueSize);
RETCODE environmentOption(long option,
                          IString& value);
```

Retrieves an environment option. Refer to the *DB2 for OS/400 SQL Call Level Interface*, SC41-4806, for valid values for option and value.

The parameters are the following:

| | |
|---|---|
| *option* | (in) Specifies which option to get |
| *value* | (out) Refers to that variable that is to contain the option's value |
| *valueSize* | (in) The size in bytes of the storage allocated for value |

***API used:*** SQLGetEnvAttr

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

### getStatusInformation

```
virtual RETCODE getStatusInformation(IString sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned SQL_SUCCESS_WITH_INFO. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns SQL_NO_DATA_FOUND. If the last operation returned SQL_SUCCESS or threw an exception, SQL_NO_DATA_FOUND is returned.

The parameters are the following:

| | |
|---|---|
| *sqlState* | (out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long. |
| *nativeError* | (out) User-allocated storage to receive the native error value. |

| | |
|---|---|
| *messageText* | (out) User-allocated storage to receive the error text. |
| *messageTextLength* | (in) Length of *messageText*. |
| *bytesAvailable* | (out) Length of text available to return. |

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND

***API used:*** SQLError

***Exceptions***

- IC4SQLException

## rollbackDatabases

```
virtual RETCODE rollbackDatabases();
```

Rolls back all open transactions across all connected IC4SQLDatabase objects.

***API used:*** SQLTransact

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## setEnvironmentOption

```
RETCODE setEnvironmentOption(long option,
                             signed long value);
RETCODE setEnvironmentOption(long option,
                             char* value,
                             long valueSize);
RETCODE setEnvironmentOption(long option,
                             IString value);
```

Sets an environment option. Refer to the *DB2 for OS/400 SQL Call Level Interface*, SC41-4806, for valid values for *option* and *value*.

The parameters are the following:

| | |
|---|---|
| *option* | (in) Specifies which option to get |
| *value* | (in) The value to which to set the option |
| *valuesize* | (in) The size in bytes of the storage allocated for value |

***API used:*** SQLGetEnvAttr

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## IC4SQLResultSet

The `IC4SQLResultSet` class represents a result set generated by an SQL SELECT statement or by a catalog function of the `IC4SQLDatabase` class. Member functions are available for the following:

- Bind and unbind of result columns to user variables or row objects
- Setting user variables from column data from a fetched row
- Retrieve the cursor name, column attributes or number of columns
- Close the result set and deallocate resources

### Derivation

`IC4SQLResultSet` does not derive from another class.

### Header File

`ic4sqrs.hpp`

## Constructors

**`IC4SQLResultSet`**`();`

## Public Members

### bindColumn

This member function associates (binds) a user variable to a column (field) in a result set. This enables data to be transferred from the result set to the application when a fetch is done.

```
virtual RETCODE bindColumn(unsigned short column,
                           short cType,
                           void* variable,
                           unsigned long variableLength,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column to bind (1-based). |
| *cType* | (in) The C language data type for the column. Data conversion can be done by specifying a type for this parameter that is different than the type of the SQL table column. |

| | |
|---|---|
| | Conversion from SQL_C_NUMERIC to SQL_CHAR is not supported. Conversion from SQL_C_DECIMAL to SQL_CHAR is not supported. |
| *variable* | (out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done. |
| *variableLength* | (in) Length of data that can be returned in the variable. |
| *bytesAvailable* | (out-deferred) Length of data returned after a fetch is done. This value could be SQL_NULL_DATA. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a NULL value. |

***Exceptions***

- IC4SQLException

To have the column type determined by the type of the user variable provided, use one of these versions:

```
virtual RETCODE bindColumn(unsigned short column,
                           short& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           long& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           double& variable,
                           unsigned long* bytesAvailable = NULL);
virtual RETCODE bindColumn(unsigned short column,
                           float& variable,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column to bind. |
| *variable* | (out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done. |
| *bytesAvailable* | (out-deferred) Size of data returned after a fetch is done. This value could be SQL_NULL_DATA. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a NULL value. |

***Exceptions***

- IC4SQLException

To bind a column to a character buffer, use this version:

```
virtual RETCODE bindColumn(unsigned short column,
                           char* variable,
                           long variableLength,
                           unsigned long* bytesAvailable = NULL);
```

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column to bind. |
| *variable* | (out-deferred) Variable to bind to column. When a fetch is done, the data for the column is placed into this variable. The variable storage must still be allocated at the time the fetch is done. |
| *variableLength* | (in) Maximum length of data that can be returned in the variable. |
| *bytesAvailable* | (out-deferred) Size of data returned after a fetch is done. This value could be SQL_NULL_DATA. The variable storage must still be allocated at the time the fetch is done. A run time error occurs if the default is used and the column has a NULL value. |

To bind a column to a variable object, use this version:

```
virtual RETCODE bindColumn(const IC4SQLVariable& variable);
```

The input variable object must contain a non-zero position number, indicating which column is to be bound to the variable. If the column is already bound, the previous binding is released.

The parameters are the following:

| | |
|---|---|
| *variable* | (in) A variable object that describes a user variable to bind. |

***API used:*** SQLBindCol

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## bindColumns

```
virtual RETCODE bindColumns(const IC4SQLVariableList& variableList);
```

Binds multiple variables to columns using an IC4SQLVariableList. Each variable object in the list must contain a non-zero position number, indicating which column is to be bound to the variable. If the column is already bound, the previous binding is released.

The parameters are the following:

*variablelist* (in) The list of user variables.

***API used:*** SQLBindCol

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## bindToRow

```
virtual RETCODE bindToRow(IC4SQLRow& row);
```

Binds each column of the result set to a row. The row object manages the storage for each column as needed. If any columns were bound previously, the previous binding is released.

The parameters are the following:

*row* (in) The row object to which columns will be bound.

***API used:*** SQLBindCol

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

## close

```
virtual RETCODE close();
```

Closes the SQL result set object.

***API used:*** SQLFreeStmt

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## IC4SQLResultSet

### columnAttribute

```
virtual RETCODE columnAttribute(unsigned short column,
                                unsigned short attributeType,
                                char* characterAttribute,
                                unsigned short characterAttributeLength,
                                unsigned short& bytesAvailable,
                                long& integerAttribute);
```

Retrieves one attribute of one column. This member function can also be used to determine the number of columns by specifying 0 as the column number.

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column number (1 is leftmost). |
| *attributeType* | (in) Type of information. For example, SQL_COLUMN_NAME. |
| *characterAttribute* | (out) User-allocated storage in which a string column attribute is returned. |
| *characterAttributeLength* | (in) Length of *characterAttribute*. |
| *bytesAvailable* | (out) Actual length of the data. If *bytesAvailable* is larger than *characterAttributeLength*, then truncation occurred. |
| *integerAttribute* | (out) User-allocated storage in which a numeric column attribute is returned. |

***API used:*** SQLColAttributes

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

### cursorName

```
virtual RETCODE cursorName(char* cursorName,
                           short cursorNameLength,
                           short& bytesAvailable);
virtual RETCODE cursorName(IString& cursorName);
```

Returns the name of the cursor associated with this result set.

The parameters are the following:

| | |
|---|---|
| *cursorName* | (out) User-allocated storage to receive the cursor name. |
| *cursorNameLength* | (in) Length of *cursorName*. |
| *bytesAvailable* | (out) Length of the cursor name available to return. If *bytesAvailable* is larger than *cursorNameLength*, then truncation occurred. |

***API used:*** SQLGetCursorName

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## describeColumn

```
virtual RETCODE describeColumn(unsigned short column,
                               char* columnName,
                               unsigned short columnNameLength,
                               unsigned short& bytesAvailable,
                               short& sqlType,
                               unsigned long& precision,
                               short& scale,
                               short& nullable);
virtual RETCODE describeColumn(unsigned short column,
                               IString& columnName,
                               short& sqlType,
                               unsigned long& precision,
                               short& scale,
                               short& nullable);
```

Returns the result descriptor information (column name, type, precision, for example) for the indicated column.

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column number |
| *columnName* | (out) String to receive the column name. |
| *columnNameLength* | (in) Length of *columnName*. |
| *bytesAvailable* | (out) Length of column name available to return. If *bytesAvailable* is larger than *columnNameLength*, then truncation occurred. |
| *sqlType* | (out) SQL datatype to which the user value should be converted. |
| *precision* | (out) The precision of the column. |
| *scale* | (out) For numeric columns, the maximum number of decimal positions. |
| *nullable* | (out) Whether NULLs are allowed for this field. Possible values are SQL_NO_NULLS or SQL_NULLABLE. |

***API used:*** SQLDescribeCol

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

## extendedFetch

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## fetch

```
virtual RETCODE fetch();
```

Fetches the next row.

***API used:*** SQLFetch

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING
- SQL_NO_DATA_FOUND

***Exceptions***

- IC4SQLException

## getData

Gets data for a fetched column. Stores (and converts if necessary) the data for the column into the variable.

If the columns have been bound to a user variable by using one of the bindColumn() methods, getData() will fail and an exception will be thrown.

```
virtual RETCODE getData(unsigned short column,
                        short cType,
                        void* value,
                        unsigned long valueLength,
                        unsigned long* bytesAvailable);
```

The parameters are the following:

| | |
|---|---|
| *column* | (in) Column number |
| *cType* | (in) The C language type |
| *value* | (out) Buffer to receive the data. |
| *valueLength* | (in) Length of *value* |

*bytesAvailable* (out) Actual length of the data available to return. If *bytesAvailable* is larger than *valueLength*, then truncation occurred.

When some parameters can be determined by the type of the variable, use one of the following versions:

```
virtual RETCODE getData(unsigned short column,
                        short& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        long& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        double& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        float& variable,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        char* variable,
                        unsigned long max,
                        unsigned long* bytesAvailable =NULL);
virtual RETCODE getData(unsigned short column,
                        IString& variable,
                        unsigned long* bytesAvailable =NULL);
```

The parameters are the following:

*column* (in) Column to retrieve.
*variable* (out) Variable to receive the data.
*max* (in) Length of *variable*.
*bytesAvailable* (out) Length of data returned in *variable* (could be SQL_NULL_DATA).

To perform `getData()` against all variables in a variable list, use this version:

```
virtual RETCODE getData(const IC4SQLVariableList& varlist);
```

Each variable object in the variable list must contain a non-zero column number. The data from that column is returned in the variable.

***API used:*** SQLGetData

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

***Exceptions***

- IC4SQLException

## getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& bytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned SQL_SUCCESS_WITH_INFO. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns SQL_NO_DATA_FOUND. If the last operation returned SQL_SUCCESS or threw an exception, SQL_NO_DATA_FOUND is returned.

The parameters are the following:

| | |
|---|---|
| *sqlState* | (out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long. |
| *nativeError* | (out) User-allocated storage to receive the native error value. |
| *messageText* | (out) User-allocated storage to receive the error text. |
| *messageTextLength* | (in) Length of messageText. |
| *bytesAvailable* | (out) Length of text available to return. |

***API used:*** SQLError

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND

## moreResults

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## numberResultColumns

```
virtual RETCODE numberResultColumns(unsigned short& columns);
```

Retrieves the number of columns in the result set.

***API used:*** SQLNumResultCols

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

## operator<<

```
ostream& operator<<(ostream& strm, IC4SQLResultSet& curs);
```

Writes the specified result set as a string to the specified stream.

The parameters are the following:

| | |
|---|---|
| *stream* | (in) The stream to write to |
| *curs* | (in) The result set object to write |

***Exceptions***

- None

## parameterData

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## putData

```
virtual RETCODE putData(const void* value,
                        long int length);
```

Use this member function with parameterData() to supply column data at statement execution time when setPosition() has been invoked.

The parameters are the following:

| | |
|---|---|
| *value* | (in) Parameter data being provided. |
| *length* | (in) Length of parameter data. |

***API used:*** SQLPutData

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

## IC4SQLResultSet

### rowAsString

```
virtual RETCODE rowAsString(IString& string);
virtual IString rowAsString();
```

Formats the current row as a string by performing a `getData()` on all the columns and building a string. Any columns that were bound are omitted from the resulting string. If a data mapping error has occurred for a column, "++++" is substituted for the column's value in the string.

***API used:*** SQLGetData

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NO_DATA_FOUND`

***Exceptions***

- IC4SQLException

### rowsAffected

```
virtual RETCODE rowsAffected(long& rows);
```

Returns the number of rows in a result set that were affected by an option of update, insert, or delete.

The parameter is the following:

*rows* (out) The number of rows in a result set that were affected.

***API used:*** SQLRowCount

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

### setPosition

This function is not supported. An `IC4SQLException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### unbind

```
virtual RETCODE unbind();
```

Unbinds all bound columns.

***API used:*** SQLFreeStmt

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

---

## IC4SQLRow

The IC4SQLRow class allows a program to fetch and save an entire row of data and perform formatting functions on the contents of the row.

### Derivation

IC4SQLRow does not derive from another class.

### Header File

ic4sqrow.hpp

## Constructors

```
IC4SQLRow();
IC4SQLRow(IC4SQLRow const & row);
```

## Public Members

### asString

```
IString asString();
```

Returns a formatted string version of the row data. If a data mapping error has occurred for a column, "++++" is substituted for the column's value in the string.

### columnData

```
const void* columnData(short column,
                       short& cType,
                       long& bytesAvailable);
```

A void pointer to the actual data as stored. If the row was never fetched into, then NULL is returned. If a data mapping error has occurred for the column, an IC4SQLException object is thrown that returns 1 from errorId() and contains 1405 in the first line of text.

The parameters are the following:

| | |
|---|---|
| *column* | (in) The number of the column for which data should be returned. |
| *cType* | (out) The C language data type of the data. For example, SQL_C_CHAR. |
| *bytesAvailable* | (out) The actual size of the data. |

***Return Value:*** A void pointer to the actual data as stored. If the row was never fetched into, then `NULL` is returned.

***Exceptions***

- IC4SQLException

## columnDataAsString

```
IString columnDataAsString(short column);
```

The parameter is the following:

| | |
|---|---|
| *column* | (in) The column number. |

Returns the data of one column stored in the row as a string. If a data mapping error has occurred for the column, an `IC4SQLException` object is thrown that returns 1 from `errorId()` and contains 1405 in the first line of text.

***Exceptions***

- IC4SQLException

## numberOfColumns

```
long numberOfColumns() const;
```

Returns the number of elements in the row.

## operator=

```
IC4SQLRow& operator=(const IC4SQLRow& row);
```

Assignment operator.

The parameter is the following:

| | |
|---|---|
| *row* | The IC4SQLRow object being assigned from. |

## operator<<

```
ostream& operator<<(ostream& stream, const IC4SQLRow& row);
```

Writes the specified row as a string to the specified stream.

The parameters are the following:

| | |
|---|---|
| *stream* | (in) The stream to write to |
| *row* | (in) The row to write |

***Exceptions***

- None

---

## IC4SQLStatement

The `IC4SQLStatement` class represents an SQL statement. This class handles the preparation and execution of SQL statements. This includes the following:

- Set parameters (bind a parameter marker to a user variable)
- Prepare an SQL text string
- Execute a prepared SQL text
- Execute an SQL text directly, without preparation
- Set and retrieve statement attributes that affect the behavior of statements and the result sets that are generated.

### Derivation

`IC4SQLStatement` does not derive from another class.

### Header File

`ic4sqst.hpp`

## Constructors

```
IC4SQLStatement(IC4SQLDatabase& database);
IC4SQLStatement(IC4SQLDatabase& database,
                const char* statement);
```

The parameters are the following:

| | |
|---|---|
| *database* | (in) The database object with which this statement is associated. The database object must have had a `connect()` done. |
| *statement* | (in) The text of the SQL statement to be prepared. |

You can construct objects of this class in the following ways:

- Create a statement without SQL statement text.
- Create a statement and prepare the statement with the specified SQL statement text.

***API***

- SQLAllocStmt

***Exceptions***

- IC4SQLException

## IC4SQLStatement

## Public Members

### bindParameter

Binds a user variable to a statement. This associates a variable with a parameter marker (indicated by '?') in an SQL statement. When the statement is executed, the value of the bound variable is used as the parameter value. This member function can also be used to specify any required data conversion. If another variable was bound to this parameter prior to this member function call, that binding is removed. Several overloaded versions of this member function are provided.

```
virtual RETCODE    bindParameter( short            pnumber,
                                  short              cType,
                                  short              sqlType,
                                  unsigned long      precision,
                                  short              scale,
                                  void  *            value,
                                  unsigned long  *   valueLength);
```

The parameters are the following:

| | |
|---|---|
| *pnumber* | (in) Number of parameter marker being bound. |
| *cType* | (in) C language datatype of the variable. |
| *sqlType* | (in) SQL datatype to which the variable should be converted. |
| *precision* | (in) The precision of the parameter value being bound. |
| *scale* | (in) For numeric columns, the maximum number of decimal positions. |
| *value* | (in-deferred) The variable that will contain the value of the parameter at execution time. |
| *valueLength* | (in-deferred) At execution time, the actual length of the value being provided as the parameter.<br>• If `NULL`, a null-terminated string is assumed.<br>• If `SQL_NULL_DATA`, a `NULL` parameter value is assumed.<br>• If `SQL_DATA_AT_EXEC`, it is assumed that the parameter value will be provided at execution time |

***API used:*** SQLSetParam

To bind a user variable of a particular type to a statement, use one of the following versions.

```
virtual RETCODE bindParameter(short pnumber,
                              short& value,
                              short sqlType =SQL_SMALLINT);
virtual RETCODE bindParameter(short pnumber,
                              long& value,
                              short sqlType =SQL_INTEGER);
virtual RETCODE bindParameter(short pnumber,
                              float& value,
                              short sqlType =SQL_REAL);
virtual RETCODE bindParameter(short pnumber,
                              double& value,
```

```
                                short sqlType =SQL_DOUBLE);
```

The parameters are the following:

| | |
|---|---|
| *pnumber* | (in) Number of parameter marker being bound. |
| *value* | (in) The variable being bound. |
| *sqlType* | (in) SQL datatype to which the variable should be converted. |

To bind a char* user variable to a statement, use this version:

```
virtual RETCODE bindParameter(short pnumber,
                              unsigned long precision,
                              const char* value,
                              unsigned long* length =NULL,
                              short sqlType =SQL_CHAR);
```

The parameters are the following:

| | |
|---|---|
| *pnumber* | (in) Number of parameter marker being bound. |
| *precision* | (in-deferred) The precision of the parameter value being bound. |
| *value* | (in-deferred) The variable being bound. |
| *length* | (in-deferred) At execution time, the actual length of the value being provided as the parameter.<br>• If `NULL`, a null-terminated string is assumed.<br>• If `SQL_NULL_DATA`, a `NULL` parameter value is assumed.<br>• If `SQL_DATA_AT_EXEC`, it is assumed that the parameter value will be provided at execution time. |
| *sqlType* | (in) SQL datatype to which the variable should be converted. |

To bind a user variable to a statement using an `IC4SQLVariable` object, use the following version.

```
virtual RETCODE bindParameter(const IC4SQLVariable& variable);
```

The parameters are the following:

| | |
|---|---|
| *variable* | (in) An `IC4SQLVariable` object describing the variable to bind. The variable object must contain a non-zero position value indicating the parameter number to which the variable is to be bound. |

***API used:*** SQLSetParam

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

## IC4SQLStatement

### bindParameters

```
virtual RETCODE bindParameters(const IC4SQLVariableList& varlist);
```

Binds multiple user variables to a statement using an `IC4SQLVariableList` object.

The parameters are the following:

| | |
|---|---|
| *varlist* | (in) Variable list containing variables to bind. Each variable object in the list that contains a non-zero position value is bound to the parameter with the corresponding parameter marker. Otherwise, the next parameter marker number following the last one bound from this list is used. If the first variable in the list has a position number of zero, then 1 is used. |

***API used:*** SQLSetParam

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

### cancel

```
virtual RETCODE cancel();
```

Cancel an outstanding `SQL_NEED_DATA` situation, or cancel processing on this statement object in another thread. If the server supports asynchronous processing, this function can cancel processing of an asynchronously submitted statement.

***API used:*** SQLCancel

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

### execute

```
virtual RETCODE execute();
virtual RETCODE execute(IC4SQLResultSet& results);
```

Executes a statement that was prepared previously. For a SELECT statement, the version that returns a result set must be used. If the statement is not a SELECT statement, the result set is closed.

The parameters are the following:

| | |
|---|---|
| *results* | (out) The `IC4SQLResultSet` object to receive the results of a SELECT statement. |

***API used:*** SQLExecute

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NEED_DATA`
- `SQL_STILL_EXECUTING`

***Exceptions***

- IC4SQLException

## executeDirect

Dynamically prepares and executes the specified SQL statement text.

Use the following version to execute a statement that is not a SELECT statement:

```
virtual RETCODE executeDirect(const char* statement,
                              long length =SQL_NTS);
```

Use the following version to execute a SELECT statement. If the statement is not a SELECT statement, the result set is closed.

```
virtual RETCODE executeDirect(IC4SQLResultSet& results,
                              const char* statement,
                              long length = SQL_NTS);
```

The parameters are the following:

| | |
|---|---|
| *results* | (out) The `IC4SQLResultSet` object to receive the results of a SELECT statement. |
| *statement* | (in) The SQL statement text. |
| *length* | (in) The length of the statement text. |

***API used:*** SQLExecDirect

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_NEED_DATA`
- `SQL_STILL_EXECUTING`

***Exceptions***

- IC4SQLException

## getParameterDescription

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## getStatusInformation

```
virtual RETCODE getStatusInformation(IString& sqlState,
                                     long& nativeError,
                                     IString& messageText);
virtual RETCODE getStatusInformation(char* sqlState,
                                     long& nativeError,
                                     char* messageText,
                                     short messageTextLength,
                                     short& messageTextBytesAvailable);
```

Returns status information associated with the last operation performed if the last operation returned SQL_SUCCESS_WITH_INFO. If another operation is performed with this object, the information for the previous operation can no longer be accessed. Once the information is retrieved, attempting to retrieve it again returns SQL_NO_DATA_FOUND. If the last operation returned SQL_SUCCESS or threw an exception, SQL_NO_DATA_FOUND is returned.

The parameters are the following:

| | |
|---|---|
| *sqlState* | (out) User-allocated storage to receive the value of SQLSTATE. Must be at least 5 bytes long. |
| *nativeError* | (out) User-allocated storage to receive the native error value. |
| *messageText* | (out) User-allocated storage to receive the error text. |
| *messageTextLength* | (in) Length of messageText. |
| *messageTextBytesAvailable* | (out) Length of text available to return. |

***API used:*** SQLError

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND

***Exceptions***

- IC4SQLException

## moreResults

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## numberParameters

This function is not supported. An IC4SQLException object will be thrown that returns 1 from errorId() and contains 0003 in the first line of text.

## parameterData

```
virtual RETCODE parameterData(void* * value);
```

Use this member function with `putData()` to supply parameter data at statement execution time. It returns the value that is used to find the data that will be provided as a parameter on an `execute()`.

The parameters are the following:

| | |
|---|---|
| *value* | (out) Pointer to the pointer to the user buffer specified on `setParameter()`. |

***API used:*** SQLParamData

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`
- `SQL_NEED_DATA`

***Exceptions***

- IC4SQLException

## prepare

```
virtual RETCODE prepare(const char* statement,
                        long length =SQL_NTS);
```

Prepares a statement.

The parameters are the following:

| | |
|---|---|
| *statement* | (in) SQL statement text. |
| *length* | (in) Length of statement text. The default value indicates that the string is null terminated. |

***API used:*** SQLPrepare

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`
- `SQL_STILL_EXECUTING`

***Exceptions***

- IC4SQLException

## IC4SQLStatement

### putData

```
virtual RETCODE putData(const void* value,
                        long int length);
```

Use this member function with `parameterData()` to supply parameter data at statement execution time.

The parameters are the following:

| | |
|---|---|
| *value* | (in) Parameter data being provided. |
| *length* | (in) Length of parameter data. |

***API used:*** SQLPutData

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_STILL_EXECUTING

***Exceptions***

- IC4SQLException

### rowsAffected

```
virtual RETCODE rowsAffected(long& rows);
```

Returns the number of rows in a table that were affected by an UPDATE, INSERT, or DELETE SQL statement executed against a table.

The parameter is the following:

| | |
|---|---|
| *rows* | (out) The number of rows in a table that were affected. 0 is returned if the previous operation was not an UPDATE, INSERT, or DELETE. |

***API used:*** SQLRowCount

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

### setCursorName

```
virtual RETCODE setCursorName(const char* cursorName,
                              short length =SQL_NTS);
```

Sets a cursor name to be associated with the next result set that is generated. If the application does not set a cursor name, the database generates a name. If the application sets a cursor name and generates a result set, and then this statement

object is used to generate another result set before closing the first result set, a different cursor name must be set; otherwise an exception is thrown.

The parameters are the following:

| | |
|---|---|
| *cursorName* | (in) The cursor name |
| *length* | (in) Length of cursor name. The default value indicates that the string is null terminated. |

***API used:*** SQLSetCursorName

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

## setParameterOptions

This function is not supported. An `IC4SQLException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

## setStatementOption

```
virtual RETCODE setStatementOption(unsigned short option,
                                   signed long value);
```

Sets a statement option.

The parameters are the following:

| | |
|---|---|
| *option* | (in) The option that is to be set. |
| *value* | (in) Value of option. |

***API used:*** SQLSetStmtOption

***Return Values***

- `SQL_SUCCESS`
- `SQL_SUCCESS_WITH_INFO`

***Exceptions***

- IC4SQLException

## statementOption

```
virtual RETCODE statementOption(unsigned short option,
                                unsigned long& value);
```

Get the current value of a statement option.

The parameters are the following:

| | |
|---|---|
| *option* | (in) The option for which the value is to be returned. |
| *value* | (out) Value of option. |

***API used:*** SQLGetStmtOption

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## unbindParameters

```
virtual RETCODE unbindParameters();
```

Unbinds all parameters previously set for this statement.

***API used:*** SQLFreeStmt

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## unprepare

```
virtual RETCODE unprepare();
```

Puts a statement back into an unprepared state, freeing up AS/400 resources. All other attributes of the statement remain unchanged.

***API used:*** SQLFreeStmt

***Return Values***

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO

***Exceptions***

- IC4SQLException

## IC4SQLVariable

The `IC4SQLVariable` class represents the information needed to bind a column, set a variable, and get column data after a row is fetched. Objects of this class can be used with an `IC4SQLVariableList` object to simplify passing a set of variables as a single parameter to certain SQL member functions.

### Derivation

`IC4SQLVariable` does not derive from another class.

### Header File

`ic4sqva.hpp`

## Constructors

The following versions are provided to allow defaulting parameters. If *position* is allowed to default to 0 and the variable is added to a variable list object, the position is set to the next position in the list following the previously added variable. If not added to a variable list, the position must be explicitly set before using the variable object to bind a parameter or column.

```
IC4SQLVariable();
IC4SQLVariable(const IC4SQLVariable& variable);
IC4SQLVariable(const IC4SQLVariable& variable,
               short position);
```

These constructors allow requesting data conversions that are different from the defaults. For example, conversion from character string to numeric, or to specify a different length or precision.

```
IC4SQLVariable(short& variable,
               short position =0,
               short sqlType =SQL_SMALLINT,
               long* bytesAvailable =NULL);
IC4SQLVariable(long& variable,
               short position =0,
               short sqlType =SQL_INTEGER,
               long* bytesAvailable =NULL);
IC4SQLVariable(float& variable,
               short position =0,
               short sqlType =SQL_REAL,
               long* bytesAvailable =NULL);
IC4SQLVariable(double& variable,
               short position =0,
               short sqlType =SQL_DOUBLE,
               long* bytesAvailable =NULL);
IC4SQLVariable(char* variable,
               unsigned long precision =0,
               short position =0,
               short sqlType = SQL_CHAR,
               long* bytesAvailable = NULL);
```

## IC4SQLVariable

The parameters are the following:

| | |
|---|---|
| *variable* | (in) The user variable to be bound to a parameter marker or to a column in a result set. Your program passes the variable to `bindParameter()` or `bindColumn()` later. The following parameters are ignored when binding a column:<br>• *sqlType*<br>• *precision* |
| *position* | (in) The position of the parameter or column to bind. |
| *cType* | (in) The C language type is determined by the type of the user variable provided. |
| *sqlType* | (in) The SQL type of the parameter or column with which this user variable is to be associated. |
| *precision* | (in) The precision of the variable. |
| *length* | (in) When used to bind a column, this is the length of the column variable. When used to bind a parameter, this is the actual length of the parameter variable or SQL_NTS if the string is null-terminated. |
| *bytesAvailable* | (in) Pointer that is later passed to `bindParameter()` or `bindColumn()` to specify a deferred output location. |

## Public Members

### assignment operator

```
IC4SQLVariable& operator=(const IC4SQLVariable& var);
```

Refers to a variable object.

### cType

```
short cType() const;
```

Returns the value of the *cType* data member.

### maxSize

```
long maxSize() const;
```

Returns the maximum size in bytes of the variable. When targeting OS/400, this member function has no meaning.

### position

```
short position() const;
```

Returns value of the *position* data member.

### precision

```
unsigned long precision() const;
```

Returns the value of the *precision* data member.

### returnSize
```
unsigned long* returnSize() const;
```

Returns the value of the *returnSize* data member.

### scale
```
short scale() const;
```

Returns the value of the *scale* data member.

### setCType
```
IC4SQLVariable& setCType(short x);
```

Sets the value of the *cType* data member.

### setMaxSize
```
IC4SQLVariable& setMaxSize(long x);
```

Sets the maximum size in bytes of the variable. When targeting OS/400, this member function has no affect on the object.

### setPosition
```
IC4SQLVariable& setPosition(short x);
```

Sets the value of the *position* data member.

### setPrecision
```
IC4SQLVariable& setPrecision(unsigned long x);
```

Sets the value of the *precision* data member.

### setReturnSize
```
IC4SQLVariable& setReturnSize(unsigned long* x);
```

Sets the value of the *returnSize* data member.

### setScale
```
IC4SQLVariable& setScale(short x);
```

Sets the value of the *scale* data member.

### setSqlType
```
IC4SQLVariable& setSqlType(short x);
```

Sets the value of the *sqlType* data member.

## IC4SQLVariableList

### setVariable

```
IC4SQLVariable& setVariable(void* x);
```

Sets the value of the *variable* data member.

### sqlType

```
short sqlType() const;
```

Returns the value of the *sqlType* data member.

### variable

```
void* variable() const;
```

Returns the value of the *variable* data member.

---

## IC4SQLVariableList

The IC4SQLVariableList class represents a collection of IC4SQLVariable objects that can be passed as a single parameter to certain database member functions. A variable list can be used with IC4SQLStatement::bindParameters(), IC4SQLResultSet::bindColumns(), and IC4SQLResultSet::getData().

### Derivation

IC4SQLVariableList does not derive from another class.

### Header File

ic4sqvl.hpp

## Constructors

```
IC4SQLVariableList();
IC4SQLVariableList(const IC4SQLVariableList& list);
```

## Public Members

### add

```
IC4SQLVariableList& add(const IC4SQLVariable& variable);
```

Adds a variable to the list. The variable is placed at the position specified by the position attribute of the variable if it is greater than 0. Otherwise, the variable is placed in the list at the position following the most recently added variable.

The parameter is the following:

*variable* The IC4SQLVariable object that is to be added to the list.

## numberOfElements

```
virtual unsigned short numberOfElements();
```

Returns the number of elements in a variable list.

## operator=

```
IC4SQLVariableList& operator=(const IC4SQLVariableList& list);
```

Assignment operator.

The parameter is the following:

*list* The IC4SQLVariableList object that is to be copied.

## operator<<

```
IC4SQLVariableList& operator<<(const IC4SQLVariable& variable);
```

Insertion operator: adds a variable to a variable list.

The parameter is the following:

*variable* The IC4SQLVariable object that is to be added to the list.

## removeAll

```
void removeAll();
```

Removes all variables in a variable list.

## removeAt

```
IC4SQLVariableList& removeAt(unsigned short position);
```

Removes a variable from a variable list.

The parameters are the following:

*position* (in) The 1-based position of the variable to remove.

## Cursor

The IC4SQLVariableList class has a nested class, Cursor, to allow iterating through a variable list using cursor member functions identical to those provided by the Collection Class Library.

The member functions available are the following:

```
Cursor(const IC4SQLVariableList& list);
IBoolean setToFirst();
IBoolean setToNext();
IBoolean setToPrevious();
IBoolean setToLast();
```

## IC4SQLVariableList

```
IBoolean isValid() const;
void invalidate();
const IC4SQLVariable& element() const;
IBoolean operator==(const Cursor& cursor) const;
```

# Chapter 4. Data Queue Classes

Data queues exist in a library on the AS/400 and are a simple and commonly used mechanism for communication between multiple tasks or programs. They are often used for communication between a client workstation and a server job.

## IC4BaseDataQueue

The `IC4BaseDataQueue` class is an abstract base class representing common functions for all data queues on an AS/400. It provides a common interface to both keyed and nonkeyed data queues.

### Derivation

`IC4BaseDataQueue` does not derive from another class.

### Header File

`ic4basdq.hpp`

## Constructors

`IC4BaseDataQueue` is an abstract base class. Constructing an instance is not allowed.

## Public Members

### asyncRead

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### cancelRequest

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### checkData

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### clear

```
virtual void clear();
```

Removes all entries from the data queue.

***Exceptions***

- IC4DataQueueException

## IC4BaseDataQueue

### close
```
virtual void close();
```

Closes the data queue object.

***Exceptions***

- IC4DataQueueException

### convert
```
virtual IBoolean convert() const;
```

Returns True if the entry is converted between local system data format and EBCDIC when reading or writing an entry. This value has no effect when targeting OS/400 because EBCDIC is the local system data format.

### dataLength
```
virtual unsigned long dataLength() const;
```

Returns the available length of the last entry accessed using read or peek.

### destroy
```
virtual void destroy();
```

Deletes the data queue from the AS/400 and does an implicit `close()`.

***Exceptions***

- IC4DataQueueException

### force
```
virtual Force force() const;
```

Returns *Force_Yes* if the `FORCE` attribute was set when the data queue was created. This attribute specifies whether the data queue is forced to auxiliary storage when entries are sent or received. Other possible values are *Force_No*.

***Exceptions***

- IC4DataQueueException

### isKeyed
```
virtual IBoolean isKeyed() const =0;
```

Returns True if the data queue is keyed.

## isOpen

```
IBoolean isOpen() const;
```

Returns True if the data queue is open.

## maxRecordLength

```
virtual unsigned long maxRecordLength() const;
```

Returns the maximum length of an entry that can be put on the data queue.

***Exceptions***

- IC4DataQueueException

## name

```
virtual inline IString name() const;
```

Returns the fully qualified name of the data queue specified on the constructor.

## open

```
virtual void open();
```

Opens the data queue object.

***Exceptions***

- IC4DataQueueException

## operator==

```
virtual IBoolean operator==(const IC4BaseDataQueue& dataQueue) const;
```

Equality operator: returns True if the system names and the qualified data queue names are the same.

***Exceptions***

- IC4DataQueueException

## operator!=

```
virtual IBoolean operator!=(const IC4BaseDataQueue& dataQueue) const;
```

Inequality operator: returns True if the system names or the qualified data queue names are different.

***Exceptions***

- IC4DataQueueException

## IC4BaseDataQueue

### operator<

```
virtual IBoolean operator<(const IC4BaseDataQueue& dataQueue) const;
```

Less-than operator: returns True if the left operand is less than the right operand as determined by the alphabetical order of the system name and then the qualified data queue name.

***Exceptions***

- IC4DataQueueException

### operator>

```
virtual IBoolean operator>(const IC4BaseDataQueue& dataQueue) const;
```

Greater-than operator: returns True if the left operand is greater than the right operand as determined by the alphabetical order of the system name and then the qualified data queue name.

***Exceptions***

- IC4DataQueueException

### operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString) =0;
```

Insertion operator: writes an entry to the data queue. *writeString* must be null-terminated.

### operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString) =0;
```

Extraction operator: reads an entry from the data queue and places it in *readString*.

### order

```
virtual Order order() const;
```

Returns the value of the attribute that specifies the sequence in which entries are received from the data queue. This applies to `read()` and `peek()` operations. For information on *order*, see “Order” on page 71.

***Exceptions***

- IC4DataQueueException

### peek

```
virtual void peek(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL) =0;
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL) =0;
```

Reads an entry from the data queue without removing the entry from the queue.

The parameters are the following:

| | |
|---|---|
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) User-allocated storage to receive sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |

## read

```
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL) =0;
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL) =0;
```

Reads an entry from the data queue.

The parameters are the following:

| | |
|---|---|
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |

## IC4BaseDataQueue

### senderID

```
virtual IBoolean senderID() const;
```

Returns True if sender information is saved with this data queue.

***Exceptions***

- IC4DataQueueException

### setConvert

```
virtual void setConvert(IBoolean value);
```

Sets whether the entry is converted between local system data format and EBCDIC when reading or writing an entry. This function has no effect when targeting OS/400 because EBCDIC is the local system data format.

***Exceptions***

- IC4DataQueueException

### systemName

```
virtual IString systemName() const;
```

Returns the name of the AS/400 system that the program is running on.

***Exceptions***

- IC4DataQueueException

### textDescription

```
virtual IString textDescription() const;
```

Returns the text that describes this data queue.

***Exceptions***

- IC4DataQueueException

### write

```
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True) =0;
```

Writes an entry to the data queue.

The parameters are the following:

| | |
|---|---|
| *writeString* | (in) Entry to be put on the data queue. |
| *writeStringLength* | (in) Size of the writeString. If this value is 0, *writeString* must be null-terminated. |

*commit* (in) True indicates the data is to be committed on the write before this function returns. This parameter has no effect on the OS/400 server. This is an optional parameter. If not specified, the data is committed.

## Enumerations

The following describes the Force enumeration and the Order enumeration.

### Force

```
enum Force { Force_No=0, Force_Yes };
```

Use these values for the FORCE data queue attribute. The attribute specifies whether the data queue is forced to auxiliary storage when entries are written to or read from the data queue.

### Order

```
enum Order { Order_LIFO=0, Order_FIFO };
```

Use these values for the SEQ data queue attribute. This attribute specifies the sequence in which entries are received from the data queue.

---

## IC4DataQueue

The IC4DataQueue class represents an AS/400 data queue that is accessed sequentially rather than with a key.

### Derivation

```
IC4BaseDataQueue
 IC4DataQueue
```

### Header File

```
ic4dq.hpp
```

## Constructors

```
IC4DataQueue(const char* dataQueueName,
             const char* systemName =NULL);
```

```
IC4DataQueue(const IC4DataQueue& dataQueue);
```

The parameters are the following:

*dataQueueName* (in) The qualified name of the data queue, specified as a path name in the library file system. For example, /QSYS.LIB/MYLIB.LIB/MYDATAQ.DTAQ. The library can be one of the following:

- a library name
- %CURLIB%
- %LIBL%

## IC4DataQueue

*systemName* (in) The name of the AS/400 system that the program is running on. This is an optional parameter. If not specified, the current system name is used.

*dataQueue* (in) An `IC4DataQueue` object. If this object is open, then the newly constructed object is opened.

***Exceptions***

- IC4DataQueueException

## Public Members

### asyncRead

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### cancelRequest

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### checkData

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### create

```
virtual void create(unsigned long maxRecordLength =80,
                    IC4BaseDataQueue::Order order =Order_FIFO,
                    IC4BaseDataQueue::Force force =Force_No,
                    const char* authority =NULL,
                    IBoolean senderID =False,
                    const char* textDescription =NULL);
```

Creates a non-keyed data queue on the AS/400 and does an implicit `open()`. The data queue can be created in a specific library or in the current library.

The parameters are the following:

*maxRecordLength* (in) The maximum length of an entry that can be put on the data queue. This is an optional parameter. The default value is 80 bytes. The allowable values are 1-64512.

*order* (in) The sequence in which entries are received from the data queue. This is an optional parameter. For information on *order*, see “Order” on page 71.

*force* (in) Specifies if the data queue should be forced to auxiliary storage when entries are sent or received. This is an optional parameter. For information on *force*, see “Force” on page 71.

| | |
|---|---|
| *authority* | (in) The public access authority given to users who do not have specific authority to the data queue. This is an optional parameter which defaults to `*LIBCRTAUT`. Other possible values are:<br>• `*CHANGE`<br>• `*ALL`<br>• `*USE`<br>• `*EXCLUDE`<br>• `authorization list name` |
| *senderID* | (in) True indicates sender information should be saved when entries are put on the data queue. This is an optional parameter. The default is not to save sender information. |
| *textDescription* | (in) Description of the data queue. This is an optional parameter and the default is a blank string. |

***Exceptions***

- IC4DataQueueException

## isKeyed

```
virtual inline IBoolean isKeyed() const;
```

Returns false because this class represents a non-keyed queue.

## operator=

```
IC4DataQueue& operator=(const IC4DataQueue& dataQueue);
```

Assignment operator: If the data queue object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4DataQueueException

## operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString);
```

Insertion operator: writes an entry to the data queue. The string to be written must be null terminated.

***Exceptions***

- IC4DataQueueException

## operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString);
```

Extraction operator: reads an entry from the data queue and places it in *readString*.

***Exceptions***

- IC4DataQueueException

## peek

```
virtual void peek(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue without removing the entry from the queue. A subsequent `read()` or `peek()` will return the same entry.

The parameters are the following:

| | |
|---|---|
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |

***Exceptions***

- IC4DataQueueException

## read

```
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue.

The parameters are the following:

| | |
|---|---|
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |

| | |
|---|---|
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |

***Exceptions***

- IC4DataQueueException

### write

```
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
```

Writes an entry to the data queue.

The parameters are the following:

| | |
|---|---|
| *writeString* | (in) Entry to be placed on the data queue. |
| *writeStringLength* | (in) Size of the writestring. If this value is 0, the *writestring* must be null-terminated. |
| *commit* | (in) True indicates the data is to be committed on the write before this function returns. This is an optional parameter. If not specified, the data is committed. This parameter has no effect on the OS/400 server. |

***Exceptions***

- IC4DataQueueException

## Inherited Public Members

| Member | Class | Page |
|---|---|---|
| clear | IC4BaseDataQueue | 66 |
| close | IC4BaseDataQueue | 66 |
| convert | IC4BaseDataQueue | 66 |
| dataLength | IC4BaseDataQueue | 66 |
| destroy | IC4BaseDataQueue | 66 |
| force | IC4BaseDataQueue | 66 |
| isOpen | IC4BaseDataQueue | 67 |
| maxRecordLength | IC4BaseDataQueue | 67 |
| name | IC4BaseDataQueue | 67 |
| open | IC4BaseDataQueue | 67 |
| operator== | IC4BaseDataQueue | 67 |

## IC4KeyedDataQueue



## IC4KeyedDataQueue

The IC4KeyedDataQueue class represents an AS/400 data queue that is accessed using a key value. The use of a key value allows data queue entries to be processed selectively. For example, a program can access just the entries with a certain key, ignoring other entries on the data queue.

### Derivation

```
IC4BaseDataQueue
  IC4KeyedDataQueue
```

### Header File

```
ic4keydq.hpp
```

## Constructors

```
IC4KeyedDataQueue(const char* dataQueueName,
                  const char* systemName =NULL,
                  const char* key =NULL,
                  unsigned short keyLength =0);

IC4KeyedDataQueue(const IC4KeyedDataQueue& dataQueue);
```

The parameters are the following:

*dataQueueName* (in) The qualified name of the data queue specified as a path name in the library file system. For example, /QSYS.LIB/MYLIB.LIB/MYDATAQ.DTAQ. The library can be one of the following:
- a library name
- %CURLIB%
- %LIBL%

*systemName* (in) The name of the AS/400 system that the program is running on. This is an optional parameter. If not specified, the current system name is used.

*key* (in) The default key to be used for operations on this queue. This is an optional parameter. If not specified, the key must be set later or passed on member function calls. Using this parameter has the same effect as using setKey().

| | |
|---|---|
| *keyLength* | (in) Size of the key. If this value is 0, the key value must be null-terminated. The key will be padded with blanks or truncated if it is not the same length as the value specified when the data queue was created. |
| *dataQueue* | (in) An `IC4KeyedDataQueue` object. If this object is open, then the newly constructed object is opened. |

***Exceptions***

- IC4DataQueueException

## Public Members

### asyncKeyReturned

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### asyncRead

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### cancelRequest

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### checkData

This function is not supported. An `IC4DataQueueException` object will be thrown that returns 1 from `errorId()` and contains 0003 in the first line of text.

### clearByKey

```
virtual void clearByKey();
virtual void clearByKey(const char* key,
                        unsigned short keyLength =0);
```

Removes all entries from the data queue that match a key. The key can be passed explicitly, or a default value can be used. If a key is not passed, the key specified on the last `setKey()` is used to select the entries. If the key value was never set, a blank key is used.

The parameters are the following:

| | |
|---|---|
| *key* | (in) String that contains the key value to use when removing entries. |
| *keyLength* | (in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created. |

***Exceptions***

- IC4DataQueueException

## convertKey

```
IBoolean convertKey() const;
```

Returns True if the key is converted between local system data format and EBCDIC when reading or writing an entry. This value has no effect when targeting OS/400 because EBCDIC is the local system data format.

## create

```
virtual void create(unsigned long maxRecordLength =80,
                    unsigned short keyLength =10,
                    IC4BaseDataQueue::Force force =Force_No,
                    const char* authority =NULL,
                    IBoolean senderID =False,
                    const char* textDescription =NULL);
```

Creates a data queue on the AS/400 and does an implicit `open()`. The data queue can be created in a specific library or in the current library.

The parameters are the following:

*maxRecordLength* (in) The maximum length of an entry that can be put on the data queue. This is an optional parameter. The default value is 80 bytes. Allowable values are 1-64512.

*keyLength* (in) The maximum allowable length of the key field. This is an optional parameter. The default value is 10 bytes. Allowable values are 1-256.

*force* (in) Specifies if the data queue should be forced to auxiliary storage when entries are sent or received. This is an optional parameter. For information on *force*, see “Order” on page 71.

*authority* (in) The public access authority given to users who do not have specific authority to the data queue. This is an optional parameter which defaults to `*LIBCRTAUT`. Other possible values are:

- `*CHANGE`
- `*ALL`
- `*USE`
- `*EXCLUDE`
- `authorization list name`

*senderID* (in) If true, sender information will be saved when entries are put on the data queue. This is an optional parameter. The default is not to save sender information.

*textDescription* (in) Description of the data queue. This is an optional parameter. The default is a blank string.

***Exceptions***

- IC4DataQueueException

## isKeyed

```
virtual inline IBoolean isKeyed() const;
```

Returns True because this class represents a keyed data queue.

## key

```
virtual IString key() const;
```

Returns the key used for comparisons when selecting an entry. The key can be set on the constructor or using `setKey()`. If this member function is used before `open()`, the value returned matches the value specified by the last `setKey()` member function call. After `open()`, the value returned may be blank padded or truncated so that it is the same length as the value specified when the data queue was created.

## keyLength

```
virtual unsigned short keyLength() const;
```

Returns the length of the key field for the data queue.

***Exceptions***

- IC4DataQueueException

## keyReturned

```
virtual IString keyReturned() const;
```

Returns the key for the entry returned by the last successful `read()` or `peek()` operation.

***Exceptions***

- IC4DataQueueException

## open

```
virtual void open();
```

Opens the keyed data queue object.

***Exceptions***

- IC4DataQueueException

## operator=

```
IC4KeyedDataQueue& operator=(const IC4KeyedDataQueue& dataQueue);
```

Assignment operator: If the KeyedDataQueue object that is being copied is open, then this object is open when the assignment operation is complete.

***Exceptions***

- IC4DataQueueException

## operator<<

```
virtual IC4BaseDataQueue& operator<<(const char* writeString);
```

Insertion operator: writes an entry to the data queue. The data must be null-terminated. Entries are written using the key value specified on `setKey()`. If `setKey()` has not been done, a blank key is used.

***Exceptions***

- IC4DataQueueException

## operator>>

```
virtual IC4BaseDataQueue& operator>>(IString& readString);
```

Extraction operator: removes an entry from the data queue and returns it in *readString*. Sender information is not returned and this member function does not wait for an entry to appear on the data queue.

The last value specified on `setSearchType()` is used when comparing keys. If `setSearchType()` has not been done, *Search_EQ* is used. The key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used. If no entries match the key specified on `setKey`, an empty string is returned.

***Exceptions***

- IC4DataQueueException

## peek

```
virtual void peek(const char* key,
                  IString& readString,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  IString* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void peek(const char* key,
                  char* readBuffer,
                  unsigned long readBufferLength,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  char* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void peek(IString& readString,
                  long waitTime =0,
```

```
                  IString* senderInformation =NULL);
virtual void peek(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from an AS/400 data queue without removing the entry from the queue.

When a version that does not require a key is used, the key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used.

If you use a version that does not have `search` or if you specify `Search_Null`, one of the following is used:

- The value from the last `setSearchType` is used.
- If `setSearchType` was never done, `Search_EQ` is used.

The parameters are the following:

| | |
|---|---|
| *key* | (in) Key value to use for comparison when selecting an entry. |
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *keyLength* | (in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |
| *search* | (in) The comparison to use when searching for a specific key. This is an optional parameter. If `Search_Null` is passed, the value from the last `setSearchType()` is used. If `setSearchType()` has not been done, *Search_EQ* is used. |

***Exceptions***

- IC4DataQueueException

## read

## IC4KeyedDataQueue

```
virtual void read(const char* key,
                  IString& readString,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  IString* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void read(const char* key,
                  char* readBuffer,
                  unsigned long readBufferLength,
                  unsigned short keyLength =0,
                  long waitTime =0,
                  char* senderInformation =NULL,
                  SearchType search =Search_Null);
virtual void read(IString& readString,
                  long waitTime =0,
                  IString* senderInformation =NULL);
virtual void read(char* readBuffer,
                  unsigned long readBufferLength,
                  long waitTime =0,
                  char* senderInformation =NULL);
```

Reads an entry from the data queue. When a version that does not require a key is used, the key specified on the last `setKey()` is used to select the entry. If the key value was never set, a blank key is used.

If you use a version that does not have `search` or if you specify `Search_Null`, one of the following is used:

- The value from the last `setSearchType` is used.
- If `setSearchType` was never done, `Search_EQ` is used.

The parameters are the following:

| | |
|---|---|
| *key* | (in) Key value to use for comparison when selecting an entry. |
| *readString* | (out) String to receive the entry. |
| *readBuffer* | (out) User-allocated storage to receive the entry. |
| *readBufferLength* | (in) Length of *readBuffer*. If this length is less than the maximum record length for the data queue, truncation may occur. |
| *keyLength* | (in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created. |
| *waitTime* | (in) Number of seconds to wait if there are no entries on the queue. This is an optional parameter. The default is not to wait for an entry. A value of -1 indicates to wait until there is an entry on the queue. |
| *senderInformation* | (out) String to receive the sender information. The length must be at least 36 bytes. This is an optional parameter. If not specified, no sender information is returned. |

*search* (in) The comparison to use when searching for a specific key. This is an optional parameter. If `Search_Null` is passed, the value used from the last `setSearchType()` is used. If `setSearchType()` has not been done, *Search_EQ* is used.

***Exceptions***

- IC4DataQueueException

## searchType

```
virtual SearchType searchType() const;
```

Returns the value that describes the type of comparison used when searching for a specific key. For information on `SearchType`, see "SearchType" on page 85.

## setConvertKey

```
void setConvertKey(IBoolean convertKey);
```

Set whether the key is converted between local system data format and EBCDIC when reading or writing an entry. This value has no effect when targeting OS/400 because EBCDIC is the local system data format.

## setKey

```
virtual void setKey(const char* key,
                    unsigned short keyLength =0);
```

Sets the key value used for comparisons when selecting an entry.

The parameters are the following:

*key* (in) Key value to use for comparison

*keyLength* (in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created.

***Exceptions***

- IC4DataQueueException

## setSearchType

```
virtual void setSearchType(SearchType searchType);
```

Sets the value that describes the type of comparison used when searching for a specific key.

The parameters are the following:

*searchType* (in) A SearchType enumeration value. For information on `SearchType`, see “SearchType” on page 85.

### write

```
virtual void write(const char* key,
                   const char* writeString,
                   unsigned short keyLength =0,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
virtual void write(const char* writeString,
                   unsigned long writeStringLength =0,
                   IBoolean commit =True);
```

Writes an entry to the data queue. When a version that does not require a key is used, the key specified on the last `setKey()` is used to write the entry. If the key value was never set, a blank key is used.

The parameters are the following:

| | |
|---|---|
| *key* | (in) Value to use for key when putting an entry on the data queue. |
| *writeString* | (in) Entry to be put on data queue |
| *keyLength* | (in) Size of the key. If this value is 0, the key value must be null-terminated. The key is padded with blanks or truncated if it is not the same length as the value specified when the data queue was created. |
| *writeStringLength* | (in) Size of the *writestring*. If this value is 0, *writestring* must be null-terminated. |
| *commit* | (in) True if the data is to be committed on the write before this function returns. This parameter has no effect on the OS/400 server. |

***Exceptions***

- IC4DataQueueException

## Inherited Public Members

| Member | Class | Page |
|---|---|---|
| clear | IC4BaseDataQueue | 65 |
| close | IC4BaseDataQueue | 66 |
| convert | IC4BaseDataQueue | 66 |
| dataLength | IC4BaseDataQueue | 66 |
| destroy | IC4BaseDataQueue | 66 |
| force | IC4BaseDataQueue | 66 |
| isOpen | IC4BaseDataQueue | 67 |
| maxRecordLength | IC4BaseDataQueue | 67 |
| name | IC4BaseDataQueue | 67 |
| operator== | IC4BaseDataQueue | 67 |
| operator!= | IC4BaseDataQueue | 67 |

| Member | Class | Page |
|---|---|---|
| operator< | IC4BaseDataQueue | 68 |
| operator> | IC4BaseDataQueue | 68 |
| order | IC4BaseDataQueue | 68 |
| senderID | IC4BaseDataQueue | 70 |
| setConvert | IC4BaseDataQueue | 70 |
| systemName | IC4BaseDataQueue | 70 |
| textDescription | IC4BaseDataQueue | 70 |

## Enumerations

The following describes the SearchType enumeration.

### SearchType

```
enum SearchType { Search_Null=0,
                  Search_EQ,
                  Search_NE,
                  Search_GT,
                  Search_GE,
                  Search_LT,
                  Search_LE };
```

Use these values for the data queue attribute that determines how entries are selected when comparing keys. *Search_Null* indicates that the last value set by `setSearchType()` should be used.

## IC4KeyedDataQueue

# Chapter 5. User Space Class

## IC4UserSpace

The IC4UserSpace class represents an AS/400 user space object.

### Derivation

IC4UserSpace does not derive from another class.

### Header File

ic4us.hpp

## Constructors

```
IC4UserSpace(const char* userSpaceName,
             const char* systemName =NULL);

IC4UserSpace(const IC4UserSpace& userSpace);
```

The parameters are the following:

| | |
|---|---|
| *userSpaceName* | (in) The qualified name of the user space specified as a path name in the library file system. For example, /QSYS.LIB/MYLIB.LIB/MYUS.USRSPC. The library value can be one of the following:<br>• a library name<br>• %CURLIB%<br>• %LIBL% |
| *systemName* | (in) The name of the AS/400 system that the program is running on. This is an optional parameter. If not specified, the current system name is used. |
| *userSpace* | (in) An IC4UserSpace object. If this object is open, then the newly constructed object is opened. |

***Exceptions***

- IC4UserSpaceException

## Public Members

### automaticExtendibility

```
Extendibility automaticExtendibility() const;
```

Returns NotExtendible if the user space is not automatically extendible and Extendible if the user space is automatically extendible.

***Exceptions***

- IC4UserSpaceException

### changeAutomaticExtendibility

```
void changeAutomaticExtendibility(Extendibility extendibility);
```

Changes the value of the attribute that determines whether a user space can be extended automatically by the system.

The parameters are the following:

| | |
|---|---|
| *extendibility* | (in) The new value for the attribute. For information on *extendibility*, see “Extendibility” on page 93. |

***Exceptions***

- IC4UserSpaceException

### changeInitialValue

```
void changeInitialValue(char newValue);
```

Changes the value of the attribute that indicates the value used to initialize future extensions of the user space.

The parameter is the following:

| | |
|---|---|
| *newValue* | (in) The new value for the attribute. Any character is valid. For best performance use '\0'. |

***Exceptions***

- IC4UserSpaceException

### changeSize

```
void changeSize(unsigned long newSize);
```

Changes the size of the user space.

The parameter is the following:

| | |
|---|---|
| *newSize* | (in) The new size of the user space in bytes. Allowable values are 1 to 16,776,704 bytes. If the new size is smaller than the previous size, the user space is truncated. If the new size is larger than the previous size, the user space is extended and initialized as specified by the initial value attribute. |

***Exceptions***

- IC4UserSpaceException

## close

```
void close();
```

Closes the user space object.

***Exceptions***

- IC4UserSpaceException

## create

```
void create(const char* domain,
            unsigned long length,
            const char* replaceUserSpace =NULL,
            const char* extendedAttribute =NULL,
            char initialValue ='\0',
            const char* textDescription =NULL,
            const char* authority =NULL);

void create(unsigned long length,
            const char* replaceUserSpace =NULL,
            const char* extendedAttribute =NULL,
            char initialValue ='\0',
            const char* textDescription =NULL,
            const char* authority =NULL;
```

Creates a user space on the AS/400. The user space can be created in a specific library or in the current library.

The parameters are the following:

*domain* (in) The domain in which the user space is created. This is an optional parameter that defaults to *SYSTEM. Other possible values are:
- *DEFAULT
- *USER

For information about the security aspects of user space domains, see *System API Reference*.

*length* (in) The maximum length of the user space in bytes. Allowable values are 1 to 16,776,704 bytes.

*replaceUserSpace* (in) Specifies whether the user space should be replaced if it already exists. This is an optional parameter that defaults to *NO. The other possible value is *YES.

*extendedAttribute* (in) The extended attribute with which the user space should be created. This is an optional, user-defined value that can be used as an identifier for the user space. If not specified, the extended attribute is set to blank.

*initialValue* (in) The initial value given to the user space when it is created. This is an optional parameter and the default value is hexadecimal zero for optimum performance.

| | |
|---|---|
| *textDescription* | (in) Description of the user space. This is an optional parameter. The default value is a blank string. |
| *authority* | (in) The public access authority given to users who do not have specific authority to the user space. This is an optional parameter that defaults to `*LIBCRTAUT`. Other possible values are:<br>• `*CHANGE`<br>• `*ALL`<br>• `*USE`<br>• `*EXCLUDE`<br>• `authorization list name` |

***Exceptions***

- IC4UserSpaceException

## destroy

```
void destroy();
```

Deletes the user space from the AS/400 system, and does an implicit close() of the user space object.

***Exceptions***

- IC4UserSpaceException

## initialValue

```
char initialValue() const;
```

Returns the initial value of the user space when it is created or when the initial value was set using `changeInitialValue()`.

***Exceptions***

- IC4UserSpaceException

## isOpen

```
IBoolean isOpen() const;
```

Returns True if the user space is open.

## name

```
const IString& name() const;
```

Returns the fully qualified name of the user space specified on the constructor.

## open

```
void open();
```

Opens the user space object.

***Exceptions***

- IC4UserSpaceException

## operator=

```
IC4UserSpace& operator=(const IC4UserSpace& userSpace);
```

Assignment operator: If the user space being copied is open, then this user space is open when the assignment operation is complete.

***Exceptions***

- IC4UserSpaceException

## operator==

```
IBoolean operator==(const IC4UserSpace& userSpace) const;
```

Equality operator: returns True if the user space name and system name are equal.

***Exceptions***

- IC4UserSpaceException

## operator!=

```
IBoolean operator!=(const IC4UserSpace& userSpace) const;
```

Inequality operator: returns True if the user space name and system name are not equal.

***Exceptions***

- IC4UserSpaceException

## operator<

```
IBoolean operator<(const IC4UserSpace& userSpace) const;
```

Less-than operator: returns True if the left operand is less than the right operand as determined by the alphabetical order of the system name and then the qualified user space name.

***Exceptions***

- IC4UserSpaceException

## operator>

```
IBoolean operator>(const IC4UserSpace& userSpace) const;
```

Greater-than operator: returns True if the left operand is greater than the right operand as determined by the alphabetical order of the system name and then the qualified user space name.

***Exceptions***

- IC4UserSpaceException

## read

```
void read(IString& readData,
          unsigned long readLength =1,
          unsigned long startingPosition =1);
void read(char* readData,
          unsigned long readLength,
          unsigned long startingPosition =1);
```

Reads the contents of the user space for the specified length.

The parameters are the following:

| | |
|---|---|
| *readData* | (out) String to receive the contents of the user space. |
| *readLength* | (in) The number of bytes to read from the user space. |
| *startingPosition* | (in) The 1-based position at which to start reading the user space. This is an optional parameter with a default value of 1. |

***Exceptions***

- IC4UserSpaceException

## size

```
unsigned long size() const;
```

Returns the size of the user space in bytes.

***Exceptions***

- IC4UserSpaceException

## systemName

```
const IString& systemName() const;
```

Returns the name of the AS/400 system that the program is running on.

***Exceptions***

- IC4UserSpaceException

## write

```
void write(const char* writeData,
           unsigned long writeDataLength =0,
           unsigned long startingPosition =1,
           Force force =NoAuxiliaryForce);
```

Writes data to the user space.

The parameters are the following:

| | |
|---|---|
| *writeData* | (in) The value to write to the user space. |
| *writeDataLength* | (in) The length of the data to write. If this value is 0, *writeData* must be null-terminated. |
| *startingPosition* | (in) The position in the user space at which to begin writing the new data. |
| *force* | (in) Indicates whether the data should be forced to auxiliary storage. This is an optional parameter. For information on *force*, see "Force." |

***Exceptions***

- IC4UserSpaceException

## Enumerations

The following describes the Extendibility enumeration and the Force enumeration.

### Extendibility

```
enum Extendibility { NotExtendible =0, Extendible };
```

Use these values for the user space attribute that specifies whether the user space is automatically extended when the end of the user space is encountered.

### Force

```
enum Force { NoAuxiliaryForce =0,
             AsynchronousForce,
             SynchronousForce
             };
```

Use these values for the user space attribute that specifies whether the user space is forced to auxiliary storage whenever the value of the user space is changed.

## IC4UserSpace

# Appendix A. Classes and Header Files

The following table lists each Access Class Library class and the header files associated with the classes.

*Table 5. Classes and Their Respective Header Files*

| Class | Header File |
|---|---|
| IC4BaseDataQueue | IC4BASDQ.HPP |
| IC4DataQueue | IC4DQ.HPP |
| IC4KeyedDataQueue | IC4KEYDQ.HPP |
| IC4DataQueueException | IC4DQEXC.HPP |
| IC4CharacterDataArea | IC4CDA.HPP |
| IC4BaseDataArea | IC4BDA.HPP |
| IC4DecimalDataArea | IC4DDA.HPP |
| IC4LogicalDataArea | IC4LDA.HPP |
| IC4DataAreaException | IC4DAEXC.HPP |
| IC4SQLDatabase | IC4SQDB.HPP |
| IC4SQLEnvironment | IC4SQENV.HPP |
| IC4SQLRow | IC4SQROW.HPP |
| IC4SQLResultSet | IC4SQRS.HPP |
| IC4SQLStatement | IC4SQST.HPP |
| IC4SQLVariable | IC4SQVA.HPP |
| IC4SQLVariableList | IC4SQVL.HPP |
| IC4SQLException | IC4SQEXC.HPP |
| IC4UserSpace | IC4US.HPP |
| IC4UserSpaceException | IC4USEXC.HPP |
| IC4CommandProcessor | IC4CP.HPP |
| IC4CommandProcessorException | IC4CPEXC.HPP |
| IC4OS400Command | IC44CMD.HPP |
| IC4Command | IC4CMD.HPP |
| IC4CommandException | IC4CMEXC.HPP |
| IC4Exception | IC4EXCPT.HPP |
| <Not a class> | IC4SQDEF.H |

## Classes and Header Files

# Bibliography

This bibliography lists related publications that you may find helpful:

- The *CL Reference*, SC41-4722, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- The *DB2 for OS/400 Database Programming* book, SC41-4701, provides a detailed discussion of the AS/400 database organization, including information on how to create, describe, and update database files on the system. Also describes how to define files to the system using OS/400 data description specifications (DDS) keywords.
- The *DB2 for OS/400 SQL Call Level Interface*, SC41-4806, provides the information necessary for application programmers to write applications using the DB2 call level interface.
- The *IBM Access Class Library User's Guide*, SC41-4623, provides information on how to use the access class library to access AS/400 data and services when using VisualAge for C++ This book discusses accessing data and services from OS/2, OS/2 Optimized, OS/400, Windows 95, and Windows NT. The following C++ classes are covered: command, data area, database, data queue, data translation, data types, program, and user space.
- The *IBM Access Class Library for Windows Reference*, SC41-4622, provides reference information on the access class library for the Windows 95 and Windows NT platforms. This book helps you create VisualAge for C++ applications that run on Windows and access AS/400 data and services.
- The *ILE COBOL/400 Programmer's Guide*, SC09-2072, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL/400 programs on the AS/400 system. It provides programming information on how to call other ILE COBOL/400 and non-ILE COBOL/400 programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- The *ILE Concepts*, SC41-4606, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- The *ILE RPG/400 Programmer's Guide*, SC09-2074, provides information about the ILE RPG/400 programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the AS/400 system. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use AS/400 files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- The *Integrated File System Introduction*, SC41-4711, provides an overview of the integrated file system, which includes:
  - What it is
  - Why you might want to use it
  - Concepts and terminology
  - The interfaces you can use to interact with it
  - The APIs and techniques you can use in programs that interact with it
  - Characteristics of individual file systems
- The *System API Reference*, SC41-4801, provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
  - Dynamic Screen Manager
  - Files (database, spooled, hierarchical)
  - Message handling
  - National language support
  - Network management
  - Objects
  - Problem management
  - Registration facility
  - Security
  - Software products
  - Source debug
  - UNIX**-type
  - User-defined communications
  - User interface
  - Work management

## Bibliography


Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.

- The *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441, describes the run-time library functions available to C++ programs developed with VisualAge for C++ for AS/400.
- The *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442, describes the C++ language as implemented by VisualAge for C++ for AS/400, including keywords, preprocessor directives, and constructs.
- The *VisualAge for C++ for AS/400 C++ Programming Guide*, SC09-2417, describes programming techniques to write ILE C++ applications that run on AS/400. Topics include performing I/O operations; working with AS/400 files, devices, pointers, and locales; implementing interlanguage calls; using templates; handling exceptions; and porting code.
- The *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416, explains how to compile, bind, and debug C++ applications that run in the Integrated Language Environment on the AS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440, describes the classes from the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions. Also describes the Binary Coded Decimal Class Library for OS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443, shows how to create C++ programs with the many different classes of the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions, and Database Access. Also describes how to create C++ programs using the IBM Binary Coded Decimal Class Library for OS/400.

The following book contains additional information that you may find helpful:

- *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, ISBN 1-55615-658-8

# Index

## D



## E



## F



## G



## H



## I

## K



## L



## M



## N



## O

# P



# R



# S

# T



# U



# V



# W

# Reader Comments—We'd Like to Hear from You!

**AS/400 Advanced Series**
**IBM Access Class Library**
**for OS/400 Reference**
**Version 3**

**Publication No. SC41-4620-00**

**Overall, how would you rate this manual?**

| | Very Satisfied | Satisfied | Dissatisfied | Very Dissatisfied |
|---|---|---|---|---|
| **Overall satisfaction** | | | | |

**How satisfied are you that the information in this manual is:**

| | | | | |
|---|---|---|---|---|
| **Accurate** | | | | |
| **Complete** | | | | |
| **Easy to find** | | | | |
| **Easy to understand** | | | | |
| **Well organized** | | | | |
| **Applicable to your tasks** | | | | |
| **THANK YOU!** | | | | |

**Please tell us how we can improve this manual:**

_____

_____

_____

_____

_____

May we contact you to discuss your responses? __ Yes __ No
Phone: (____) _____ Fax: (____) _____ Internet: _____

**To return this form:**

- Mail it
- Fax it
  United States and Canada: **800+937-3430**
  Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

_____ _____
Name Address

_____ _____
Company or Organization

_____ _____
Phone No.

**Reader Comments—We'd Like to Hear from You!**
SC41-4620-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

# BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

SC41-4620-00

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC41-4620-00

*Spine information:*

AS/400 Advanced Series

IBM Access Class Library
for OS/400 Reference

*Version 3*