HCL Informix 14.10

HCL Informix Object Interface for C++ Programmer's Guide

Contents

Chapter 1. Informix® Object Interface for C++ Guide

The *Informix® Object Interface for C++ Programmer's Guide* describes how to develop HCL Informix® client applications by using the object-oriented C++ programming language.

The encapsulates HCL Informix® features into an easy-to-use class hierarchy and extensible object library.

The Object Interface for C++ is documented in these topics. The DataBlade® API is documented in the *Informix® DataBlade® API Programmer's Guide*. The GLS API, from which the Object Interface for C++ **ITLocale** class is derived, is documented in the *Informix® GLS API Programmer's Guide*.

These topic refer extensively to the example programs included with the Object Interface for C++.

These topics are written for two audiences:

- Developers that use C++ to create database client applications for HCL Informix® servers
- DataBlade® developers who use to create value objects that allow C++ client applications to support DataBlade® module data types

To use these topics, you must know C++. Familiarity with the Microsoft™ Component Object Model (COM) is also helpful when working with the Object Interface for C++.

All public names in the begin with IT.

For information about software compatibility, see the Informix® Client SDKrelease notes.

Architecture of the object interface for C++

The encapsulates HCL Informix® database server features into a class hierarchy.

Operation classes provide access to Informix® databases and methods for issuing queries and retrieving results. Operation classes encapsulate database objects such as connections, cursors, and queries. Operation class methods encapsulate tasks such as opening and closing connections, checking and handling errors, executing queries, defining and scrolling cursors through result sets, and reading and writing large objects.

Value interfaces are abstract classes that provide specific application interaction behaviors for objects that represent HCL Informix® database values (value objects). You can interact with your data by using extensible value objects. Built-in value objects support ANSI SQL and C++ base types and complex types such as rows and collections. You can create C++ objects that support complex and opaque data types.

Operation classes

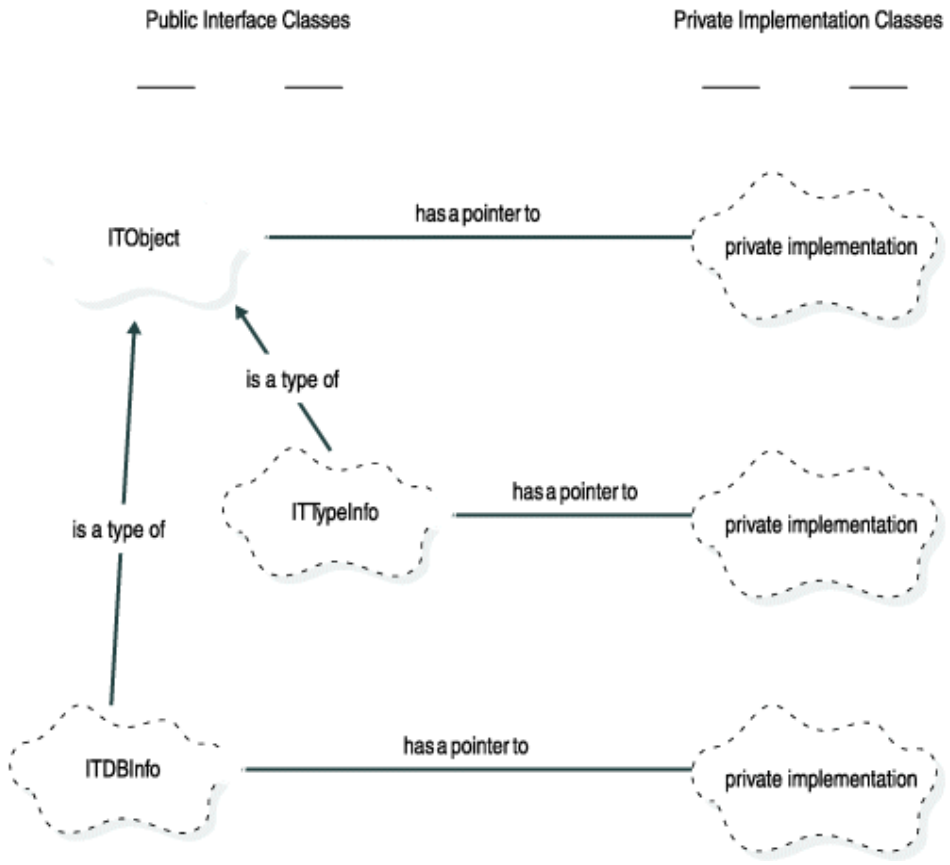
applications create instances of public operation classes, which contain pointers to private implementation classes.

Although this interface/implementation approach adds an extra level of indirection, it provides important benefits:

- Applications do not depend on the implementation of the underlying class because the implementation class is inaccessible.
- Performance of copy operations is improved because applications copy only the implementation pointer of the object and not the entire object.
- Applications can easily use automatic variables as opposed to heap-allocated variables. Automatic variables are automatically deallocated when they pass out of scope, which helps avoid memory leaks. The implementation class tracks references to objects, removing objects only when they are no longer referenced.

The following figure illustrates the relationship between the public interface classes and private implementation classes.

Figure 1. Public interface and private implementation of operation classes



The Object Interface for C++ defines the following operation classes.

Operation class	Description	See
ITConnection	Manages a database connection.	The ITConnection class on page 49
ITConnectionStamp	Maintains stamp information about a connection.	The ITConnectionStamp class on page 50

Operation class	Description	See
ITContainerIter	Extracts C++ base-type values (such as int, long, or double) from a container value object.	The ITContainerIter class on page 51
ITCursor	Defines cursors and manages results.	The ITCursor class on page 53
ITDBInfo	Stores database information.	The ITDBInfo class on page 55
ITDBNameList	Allows the user to obtain database names.	The ITDBNameList class on page 57
ITErrorManager	Provides base class functionality for managing error callbacks.	The ITErrorManager class on page 57
ITFactoryList	Adds mappings from HCL Informix® data types to functions that build value objects to represent instances of these data types.	The ITFactoryList class on page 58
ITInt8	Provides an 8-byte integer class.	The ITInt8 class on page 60
ITLargeObjectManager	Supports large objects.	The ITLargeObjectManager class on page 62
ITLocale	Provides GLS support.	Online notes
ITMVDesc	Not an operation class, but a descriptor that holds the instance information necessary to create a value object.	The ITMVDesc class on page 67
ITObject	Provides the base class for public operation class interface objects.	The ITObject class on page 67
ITPreserveData	Provides an interface for maintaining a reference to database data received from the server, for use by the implementer of a value object.	The ITPreserveData class on page 68
ITQuery	Issues SQL queries to the HCL Informix® database.	The ITQuery class on page 69
ITRoutineManager	Provides fast path execution of DataBlade® API functions.	The ITRoutineManager class on page 70
ITStatement	Provides support for the execution of prepared queries that return no rows.	The ITStatement class on page 71
ITString	Provides a string class.	The ITString class on page 74

Operation class	Description	See
ITSystemNameList	Allows the user to obtain host system names.	The ITSystemNameList class on page 76
ITTypeInfo	Stores information about database types.	The ITTypeInfo class on page 76

For detailed descriptions of these operation classes, see Operation class reference on page 49.

Value interfaces and value objects

The creates C++ objects that encapsulate data retrieved from a database. These value objects are created by the Object Interface for C++ by using an extensible class factory that maps server data types to C++ objects.

DataBlade® developers can create value objects that represent new Informix® data types. Developers can use the Object Interface for C++ to write client applications that operate with these new value objects. Object Interface for C++ client applications do not depend on the representation of the object in the database; if the database representation changes, the corresponding value object can be altered and the existing applications continue to run. Code for value objects can be compiled into an application or dynamically loaded into an application from shared libraries.

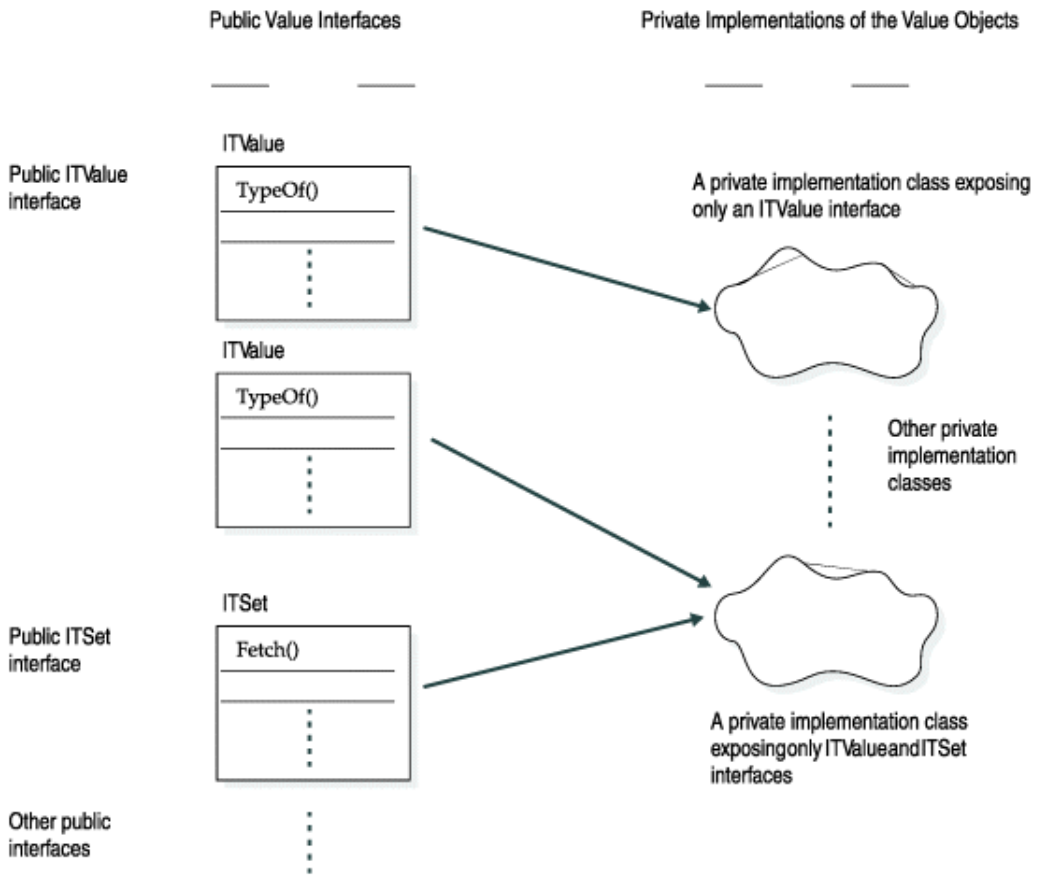
The value object design is compatible with the Microsoft™ Common Object Model (COM) in the sense that it enables objects to expose behaviors through *interfaces*. An interface is an abstract class that encapsulates the methods associated with a specific behavior.

For example, to indicate that an object can behave as a container, the object exposes the **ITContainer** interface; to indicate that an object can convert its value to a C++ base type (such as **int** or **double**), an object exposes the **ITConversions** interface; and other interfaces.

Interfaces are extracted from an object by calling a **QueryInterface()** function provided in **ITEssential**, which is the base class of all value interfaces. When the **QueryInterface()** function is called, the caller specifies the interface ID of the interface you want. If the object exposes the requested interface, then **QueryInterface()** returns `IT_QUERYINTERFACE_SUCCESS` and sets its second argument to the address of the interface you want.

The following figure illustrates the relationship of the application interface to the implementation.

Figure 2. Public interface and private implementation of value objects



The Object Interface for C++ defines the following value interfaces.

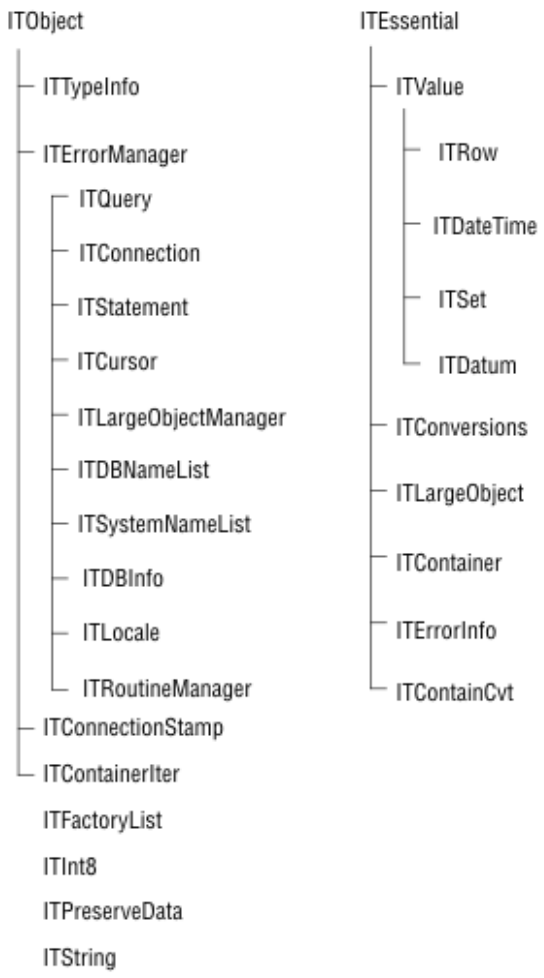
Interface	Description	See
ITContainCvt	Decomposes an object into C++ base type instances.	The ITContainCvt interface on page 79
ITContainer	Provides access to the container members.	The ITContainer interface on page 80
ITConversions	Converts data to C++ base classes or strings.	The ITConversions interface on page 80
ITDateTime	Allows access to the fields of a database date/time object.	The ITDateTime interface on page 82
ITDatum	Supports the functionality of the basic value object, including access to the underlying data.	The ITDatum interface on page 83

Interface	Description	See
ITErrorInfo	Exposes error information about objects for which invalid operations can cause server errors.	The ITErrorInfo interface on page 83
ITEssential	Serves as the base of the value interface classes.	The ITEssential interface on page 84
ITLargeObject	Manipulates a large object returned by a query.	The ITLargeObject interface on page 85
ITRow	Provides access to row values.	The ITRow interface on page 86
ITSet	Provides access to collection results.	The ITSet interface on page 87
ITValue	Supports the basic functionality of the value object.	The ITValue interface on page 87

Class hierarchy

The following diagram shows the inheritance hierarchy.

Figure 3. C++ inheritance hierarchy



Implementation notes

This section describes the programming restrictions and practices.

Restrictions

The is subject to the following restrictions:

- The Object Interface for C++ does not support object persistence for application classes; it does not automatically map instances of database tables to application classes or vice versa.
- You cannot directly update the database data by modifying the corresponding value objects; to modify the database data that corresponds to the data returned to client programs in value objects, you must issue SQL queries, or the methods ITCursor::UpdateCurrent() and ITCursor::DeleteCurrent().
- You cannot develop server functions by using the Object Interface for C++.
- Do not mix database access through the Object Interface for C++ and lower-level interfaces (like the DataBlade® API) in the same application.
- The Object Interface for C++ is not thread-safe. Do not use Object Interface for C++ in multi-threaded applications or environments.

Passing objects—compiler dependency

When you pass an object to a function by value, the C++ compiler creates a temporary copy of the object to pass to the function. The compiler deletes the object after the function returns. The exact time at which temporary objects are deleted is compiler-dependent. For this reason, your application must not rely on the automatic destruction of temporary objects.

For example, if you pass an **ITConnection** object to a function by value and start the AddCallback method on the connection inside the function, the temporary connection object (on which you added the callback) might or might not exist immediately after the function returns. Because both the original connection object and the copy refer to the same underlying server connection, the new callback might or might not remain in effect on the underlying connection when your function returns.

To ensure consistent behavior, call DelCallback inside your function when the new callback is no longer required. Do not rely on the automatic destruction of the connection object parameter by the compiler to remove the callback from the underlying server connection. For details about DelCallback, see The ITErrorManager class on page 57.

Informix® database server compatibility

The can be used to create database client applications that run against HCL Informix® databases. However, classes and methods that support version 9,x and 10.x extensibility features are not supported with version 7.x databases.

HCL Informix® version 7.x does not support the **boolean**, **int8**, **blob**, **clob**, or **lvarchar** data types or the Informix® extended data types: opaque, distinct, row, and collection.

Some of the Object Interface for C++ examples work only with Informix® version 9.x and 10.x, since the version 7.x Dynamic Server SQL parser does not support Informix® data type casting syntax (*value*::*data_type*) in SQL statements.

Object Interface for C++ dynamic loading and object delegation technique are only useful with Informix® databases.

Globalization

The provides functionality based on Informix® Global Language Support.

The **ITLocale** class, described in The ITLocale class on page 92, encapsulates the GLS API. It provides methods to perform locale-sensitive conversions between the text and binary forms of the date, time, numeric, and money data types. It also provides support for multibyte character strings and for quoted type names.

Call ITLocale::Current() to obtain a pointer to the current client locale and use ITLocale::ConvertCodeset() to convert data between the two code sets.

The **ITString** class encapsulates a string in a client locale. When a string is retrieved from a server, it is converted to the client locale. Locale-specific rules govern the following operations:

- Date/time, numeric, and money string formatting
- Error messages produced by the Object Interface for C++
- String operations such as Trim(), concatenation, and other operations

Client locale is established at the startup time of the application based on the value of the **CLIENT_LOCALE** environmental variable.

For more information, see the *Informix® GLS User's Guide*.

ITFactory list and the type map

Type names for the **ITFactoryList** constructor or in the type map file can contain any characters in the current client locale, except NULL.

Type names can contain multibyte characters. If a type name includes white space characters, enclose the type name in a pair of double quotation marks in the type map file. If the type name contains a double quotation mark character, place a double quotation mark character before it.

Type name searches in the current client locale are not case-sensitive.

Issue database queries and retrieve results

To interact with a database, your C++ client application uses the operation classes of the . These classes have methods for opening database connections, submitting queries, and manipulating database cursors. This section describes how to use these methods.

Using operation classes

The `csql.cpp` example is a small application that uses the **ITQuery** and **ITConnection** classes to provide a simple command-line interface that accepts SQL commands from the standard input, transmits the commands to the database, and the results are displayed.

The major steps of the program are as follows.

1. Open the connection.

   Before any database interaction can take place, the connection with the database must be established. Opening the connection without any arguments instructs the interface to use the default system, database, user name, and password. For details about connection defaults, see The ITDBInfo class on page 55.

```
ITConnection conn;
conn.Open();
```

2. Build an **ITQuery** object for the connection.

   ```
   ITQuery query(conn);
   ```

   A query object is used to issue database queries and to access result sets. An operation class is always created in the context of a server connection.
3. Read lines of input from **stdin** by using the C++ **iostream** library methods.

   ```
   while (cin.getline(qtext, sizeof(qtext)))
   {
   }
   ```
4. Execute the query read from **stdin** by using the **ExecForIteration** method of the query object.

   ```
   if (!query.ExecForIteration(qtext))
   {
   }
   ```
5. Loop through the result rows of the query.

   ```
   ITRow *comp;
   int rowcount = 0;
   while ((comp = query.NextRow()) != NULL)
   {
   }
   ```

   A row is extracted from the result set of a query by using the NextRow method of the query object. The code shows the declaration of a pointer to the row interface for an object that receives the result data, and the loop that reads the result data into the row object.

   This is an example of the use of a value object in the program: The NextRow method returns a pointer to an **ITRow** interface. The pointer returned by NextRow is not a pointer to an actual object; it is a pointer to an interface that is exposed by an object.
6. Print the row.

   ```
   cout << comp->Printable() << endl;
   ```

   Every value object exposing an **ITValue** or **ITValue**-derived interface supports the Printable method, which returns the object as a printable string in a constant **ITString** object. This object can be put directly on the **stdout** stream. For details about the **ITString** class, see The ITValue interface on page 87.
7. Release the row.

   ```
   comp->Release();
   ```

   The value interface returned to the application must be explicitly released by the application. A value object tracks the number of outstanding references to it, and when the last reference is released, deletes itself.
8. Close the connection.

   ```
   conn.Close();
   ```

Closing a connection deletes any saved data associated with the connection. Because a value object might hold a reference to this saved data, it must track whether the underlying data has been deletes. For details, see Value object management on page 24.

Create connections

To specify connection parameters (system, database, user name, and password) when creating a connection, your application creates an instance of the **ITDBInfo** class. If the application uses the default connection parameters, you can create a connection without the use an instance of the **ITDBInfo** class.

After an **ITDBInfo** variable is constructed, it can be used to establish multiple database connections. However, after a connection has been established by using a given **ITDBInfo**, that instance of **ITDBInfo** cannot be changed, nor can any copy of it be modified. The **ITDBInfo** instance is said to be frozen. To detect whether an **ITDBInfo** object has been frozen, use the ITDBInfo::Frozen() method.

The default user name and password are those of the current user. The default database name is the name of the current user. The default server name is specified in the UNIX™ **$INFORMIXSERVER** environment variable or in the Windows™ registry. If the **ITDBInfo** instance is not frozen, you can modify these values with the ITDBInfo::SetDatabase(), ITDBInfo::SetUser(), ITDBInfo::SetPassword(), and ITDBInfo::SetSystem() methods.

Find system names and database names

Many client applications determine what database to use at run time, sometimes allowing users to select from alternatives. You can use the **ITSystemNameList** class and the **ITDatabaseNameList** class to retrieve lists of HCL Informix® servers and databases.

The following topics describe how to use these classes.

Using ITSystemNameList

The following excerpts from `sysname.cpp` illustrate the use of **ITSystemNameList**.

1. The Create() method creates the system name list by looking into the `sqlhosts` file (on UNIX™) or from the registry entry under the `HKEY_LOCAL_MACHINE\Software\Informix\sqlhosts` key (on Windows™).

   ```
   ITSystemNameList list;
   ITBool created = list.Create();
   ```

2. The system name list is displayed by the ITSystemNameList::NextSystemName() method.

   ```
   while (ITString::Null != (current = list.NextSystemName()))
       {
       cout << current << "\tALWAYS_DIFFERENT" << endl;
       last = current;
       }
   ```

Using ITDBNameList

The following excerpts from **dbname.cpp** illustrate the use of **ITDBNameList**.

1. ITDBNameList::Create() creates an instance of **ITDBNameList** that lists the databases from the servers contained in the **DBPATH** and **INFORMIXSERVER** environment variables.

```
ITDBNameList dbnl;
ITBool created;
    created = dbnl.Create();
```

2. The database name list is displayed by the ITDBNameList::NextDBName() method.

```
void
DisplayITDBNameList(ITDBNameList &dbname)
{
    ITString str;

    cout << "Parsing the DBNameList by calling NextDBName()
        method "<< endl;

    while (ITString::Null != (str = dbname.NextDBName()))
        cout << str << "\tALWAYS_DIFFERENT" << endl ;
}
```

Manage errors

Most operations, such as issuing queries, fetching rows, and setting transaction states, return a result code that your application checks. Operations that return pointers typically return `NULL` to indicate an error. Operations that return a Boolean result typically return `FALSE` to indicate an error.

To specify a routine to be called whenever an error or warning is posted, your application can associate a callback function with an instance of these classes. If an error occurs, the callback function is executed. See The ITErrorManager class on page 57 for the callback function signature.

To check errors from operation objects, call the Error and ErrorText methods after an operation is performed, or include calls to the Error and ErrorText methods in the body of an error callback function added to the object. Within an error callback function, the only safe operations are calls to the Error, ErrorText, Warn, WarningText, and SqlState methods to examine the **ErrorManager** object.

Your own data structures can be accessed with the user data parameter, which is untouched by the . Any operations in the callback function that are performed by using the Object Interface for C++, such as calls to the operation class methods that submit queries, have undefined results.

The **ITErrorManager** base class gives its derived classes the ability to manage errors returned by the server or generated within the Object Interface for C++.

Callbacks added to an operation class derived from **ITErrorManager** are added to that interface object. If the interface object is deleted, the callbacks registered on that interface are removed. If the interface object is deleted while the implementation is still present and the callbacks were not removed, there is no valid interface object reference for the first parameter of the callback when the implementation calls the callback, and a segmentation violation might occur. The destructor of **ITErrorManager** removes such a callback.

To track all errors on a connection, set a callback function on the connection object. When processing errors from a connection object, be sure to check the return status from the operation itself, and not from the **Error** method. To track all errors for a specific object, set a callback function on the object itself.

Using the error handling feature

The `csql2.cpp` example consists of the `csql.cpp` SQL interpreter example enhanced with error-handling code. The following steps describe the error-handling features used in the `csql2.cpp` example:

1. Add the error callback function:

```
ITCallbackResult
my_error_handler(const ITErrorManager &errorobject,
                 void *userdata,
                 long errorlevel)
{
    // Cast the user data into a stream
    ostream *stream = (ostream *) userdata;
    (*stream) << "my_error_handler: errorlevel="
          << errorlevel
          << " sqlstate="
              << errorobject.SqlState()
              << ' '
              << errorobject.ErrorText()
              << endl;
    return IT_NOTHANDLED;
}
```

   The arguments to the callback function are the object on which the error appeared, a field (**userdata**) passed to the callback function, and an indicator of the severity of the error (for details about levels of errors, see The ITErrorManager class on page 57). In this example, the callback function casts the user data field into a C++ **ostream** object and prints the error text and SQL error code (the ISO standard SQLSTATE) on the output stream. The user data in the example must be an **ostream** pointer.

2. Add the callback function to the error handler list maintained by the query object:

```
query.AddCallback(my_error_handler, (void *) &cerr);
```

The following dialog shows how the **csql2.cpp** program handles an erroneous SQL statement. At the prompt (>), the user types `error;` (which is not valid SQL) and an error message is displayed by the error handler of the **csql2.cpp** program:

```
% csql2
Connection established
> error;
my_error_handler: errorlevel=2 sqlstate=42000
X42000:-201:Syntax error or access violation
Could not execute query: error;
0 rows received, Command:
>
```

Connection transaction states

A connection to a database is said to be in one of a number of *transaction states*. Transaction states show how queries submitted on the connection are committed. Some server operations can only take place within a transaction. For example, updateable cursors can only be opened within a transaction.

The **ITConnection** class is used to manage connections and includes methods to set and inquire about the transaction state. The following table lists the connection transaction states.

State	Effect of setting this state	Significance when retrieved from ITConnection
None	Not allowed to set	Not connected to a server
Auto	Not allowed to set	In auto commit mode (each SQL statement is a separate transaction)
Begin	Start a transaction	Entered or in a transaction
Commit	Commit the transaction	Last transaction was committed
Abort	Abort the transaction	Last transaction was aborted/rolled back

The `csql3.cpp` example adds transaction monitoring capabilities to the SQL interpreter example. The following steps point out the transaction monitoring features:

1. If the session is within a transaction, print `"TRANSACTION>"` as the prompt. The following code shows the use of the GetTransactionState method to check the transaction state:

```
if (conn.GetTransactionState() == ITConnection::Begin)
{
    cout << "TRANSACTION> ";
}
else
{
    cout << "> ";
}
```

2. If the session exits while it is within a transaction, stop the transaction. The data is returned to the state it was in when the transaction started. The following code shows the use of the GetTransactionState method to check the transaction state and SetTransactionState to set the state:

```
if (conn.GetTransactionState() == ITConnection::Begin)
{
    cerr << endl
         << "Exit within transaction, aborting transaction"
         << endl;
    conn.SetTransaction(ITConnection::Abort);
}
```

The output from the example is similar to the following, when the user exits after issuing a `begin work` statement:

```
% csql3
Connection established
```

```
> begin work;
0 rows received, Command:begin work
TRANSACTION> EOF
Exit within transaction, aborting transaction
```

Issue queries

There are a number of different ways to issue SQL queries in the , each suitable for different application requirements.

The following table summarizes the methods used for issuing queries.

Method	Description
ITQuery::ExecForStatus	Execute a query that does not return rows (such as CREATE, INSERT, UPDATE, or DELETE). Return a result code that says whether the query resulted in a server error.
ITQuery::ExecOneRow	Execute a query that returns one row; flush any results other than the first row. Useful for quickly submitting queries that only return a single row, such as `select count(*) from systables.`
ITQuery::ExecToSet	Execute a query and retrieve all the result rows into a saved row set managed on the client.
ITQuery::ExecForIteration	Execute a query and return one row to the application on every call to ITQuery::NextRow.
ITCursor::Prepare/ITCursor::Open	Define a cursor for a select statement and return rows to the client on calls to ITCursor::Fetch.
ITStatement::Prepare/ITStatement::Exec()	Prepare and execute a query that returns no rows.

When to use the different ITQuery methods

This section describes how to use the query methods appropriately.

The ExecForStatus method

Use the ExecForStatus method of the **Query** object for queries when the application does not need any data returned from the query (for example, DDL statements such as CREATE TABLE, DROP TABLE, CREATE VIEW, or DML statements such as UPDATE).

The ExecForStatus method returns `FALSE` if a server error occurred.

The ExecOneRow method

Use the ExecOneRow method of the **Query** object for queries that return (or are expected to return) one row.

The ExecOneRow method returns an **ITRow** interface pointer that represents the result row, or `NULL` if there is an error or if no row is returned. If the query returns more than one row, the first row is returned and the rest are discarded.

The ExecToSet method

Use the ExecToSet method of the **Query** object for queries that return more than one row.

The ExecToSet method runs the query to completion and stores the results in the memory of the client program. If the result set is large, the memory of the client might be inadequate. The results returned by ExecToSet are accessible in arbitrary order.

Using ExecToSet, the connection is checked in after the call is completed. For details about checking connections in or out, see The ITConnection class on page 49.

The ExecForIteration method

Use the ExecForIteration method of the **Query** object for queries that return a large result set that must be processed a row at a time.

After issuing the query with ExecForIteration, your application must call NextRow to access the individual rows in the result set. While your application is processing the rows returned by ExecForIteration, the connection to the database server cannot be used for another query. You can, however, free up the connection to the server by using the ITQuery::Finish method to finish query processing without retrieving all rows.

This method is the query-executing mechanism most similar to executing a **select** statement by using the DataBlade® API mi_exec() and mi_next_row calls. Also, this method does not enable nonsequential access to the rows.

Query method example

The `queryex.cpp` example demonstrates use of the ExecForStatus, ExecOneRow, and ExecToSet methods.

The following excerpts illustrate the use of the query methods in the `queryex.cpp` example:

1. Call ITQuery::ExecOneRow() to check if the table **informixfans** exists in the database. If the table does not exist, use ITQuery::ExecForStatus() to create it.

```
// Does the table exist? If not, then create it.
ITRow *r1 = q.ExecOneRow(
    "select owner from systables where tabname = 'informixfans';");
if (!r1
     && (!q.ExecForStatus(
         "create table informixfans (name varchar(128));")))
    {
    cerr << "Could not create table 'informixfans'!" << endl;
```

```
    return 1;
    }
```

2. Call ITQuery::ExecToSet to fetch the results of a **select** statement:

```
// Show the contents of the table
cout << "These are the members of the Informix fan club, version ";
ITValue *rel = q.ExecOneRow
    ("select owner from systables where tabname = ' VERSION';");
cout << rel->Printable() << " ALWAYS_DIFFERENT" << endl;
rel->Release();

ITSet *set = q.ExecToSet
    ("select * from informixfans order by 1;");
if(!set)
{
    cout << "Query failed!" << endl;
    conn.SetTransaction(ITConnection::Abort);
    conn.Close();
    return -1;
}
ITValue *v;
while ((v = set->Fetch()) != NULL)
{
    cout << v->Printable() << endl;
    v->Release();
}
set->Release();
```

Using prepared statements

Prepared statements can be used to perform INSERT, UPDATE, and DELETE functions efficiently and to pass binary data as parameters. The encapsulates prepared statement functionality in the **ITStatement** class.

The following excerpts illustrate the use of the `loadtab.cpp` example to load a table from a text file by using a prepared statement.

1. To use a prepared statement, the application creates an instance of **ITStatement** on the opened connection.

```
ITStatement stmt(conn);
```

2. The application prepares the SQL statement, which creates the statement parameters.

```
if(!stmt.Prepare(sql))
    return -1;
```

   Created parameters have the value `NULL`.

3. When the application must set a parameter value, it obtains the **ITValue*** of the parameter through the call to the Param() function.

```
ITValue *param = stmt.Param(paramno);
```

   The application can call the NumParams() function to obtain the number of parameters.

4. The application sets the parameter value by using ITValue::FromPrintable(), or it obtains the required interface by calling the QueryInterface() function and uses its update routines.

```
if (!param->FromPrintable(pdb))
    {
    cerr << "Could not set parameter "
         << paramno << " to '" << pdb << "'" << endl;
    return -1;
```

The application must release the **ITValue** interface of the parameter by calling param->Release().

5. After all parameter values are set, the application executes the prepared query.

```
if (!stmt.Exec())
{
    cerr << "Could not execute statement" << endl;
    return -1;
```

The application can use the RowCount() function to determine the number of rows affected by the last query executed. The application can then reset the parameter values and re-execute the query. Any parameter values that have not been reset stay the same.

After the application is completed work with the prepared statement, it drops the statement by using the Drop() function.

The same instance of **ITStatement** can be used to prepare another SQL statement by calling **Prepare()**, which calls **Drop()** for any currently prepared statement.

Using cursors

Cursors can be used to efficiently perform SELECT statements with parameters and to pass binary data as parameters. Cursors can also be used to update database tables. The encapsulates cursor functionality into the **ITCursor** class.

The following excerpts from the `cursupd.cpp` example illustrate the use of **ITCursor**.

1. To use a cursor, the application creates an instance of **ITCursor** on the opened connection.

```
ITCursor cursor(conn);
```

2. The cursor is opened in a transaction. The preparation of the SELECT statement creates statement parameters.

```
conn.SetTransaction(ITConnection::Begin);

if(!cursor.Prepare("select b from bar where b < ?::integer;"))
{
```

If the application does not specify a parameter type name list, default parameter types are used (see The ITStatement class on page 71). Created parameters have NULL values.

3. When the application must set a parameter value, it obtains the **ITValue *** of the parameter through the call to the Param() function.

```
ITValue *par = cursor.Param(0);
if(!par)
```

The application can call the NumParams() function to obtain the number of parameters.

4. The application sets the parameter value by using ITValue::FromPrintable().

```
if(!par->FromPrintable("3"))
{
```

Alternatively, the application can obtain the required interface by calling QueryInterface() and use the update functions provided by the interface.

5. After all parameter values are set, the application opens the cursor with the flags representing the sum of ITCursor::Flags values.

```
if(!cursor.Open(0, "bar"))
{
```

By default, the cursor is opened as updateable and nonscrollable. The cursor cannot be opened as updateable and scrollable at the same time. If the application uses the UpdateCurrent() or DeleteCurrent() functions of the cursor, it must provide the name of the table that the cursor is created on as a second argument of Open().

6. The application can use a fetch function to find the row from the cursor. The fetch function accepts a pointer to the outer unknown interface for delegation (for more details about delegation, see Object Containment and Delegation on page 44). The pointer is null by default.

The fetch function can perform the positional fetch. If the cursor was not opened as scrollable, positional fetch fails. The application can call the IsScrollable() function to check whether the cursor is scrollable. The fetch function returns the pointer to the **ITValue** interface of the retrieved row. The NextRow() function returns the pointer to the **ITRow** interface of that row.

```
ITRow *row;
while(row = cursor.NextRow())
    {
    ITValue *col = row->Column(0);
    if(!col)
        {
        cerr << "Couldn't get the column from the cursor's row" << endl;
        return -1;
        }

    cout << "Column 0 was " << col->Printable() << endl;
```

The following excerpts from the `curstst.cpp` example program illustrate the use of a scrollable cursor.

a. Fetch rows from the beginning to the end of the result set.

```
cout << "FORWARDS" << endl;
while ((rowValue = cursor.Fetch()) != NULL)
{
    rowcount++;
    cout << rowValue->Printable() << endl;
    rowValue->Release();
}
```

b. Fetch rows from the end to the beginning of the result set.

```
cout << "BACKWARDS" << endl;
for (;;)
{
    if (!(row = cursor.NextRow(0, ITPositionPrior)))
        break;
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

c. Fetch every second row from the beginning to the end of the result set.

```
cout << "EVERY SECOND" << endl;
for (;;)
{
    if (!(row = cursor.NextRow(0, ITPositionRelative, 2 )))
        break;
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

d. Fetch the third row from the result set.

```
cout << "THIRD" << endl;
row = cursor.NextRow(0, ITPositionAbsolute, 3);
if (row != NULL)
{
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

e. Fetch the first row of the result set.

```
cout << "FIRST" << endl;
row = cursor.NextRow(0, ITPositionFirst);
if (row != NULL)
{
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

f. Fetch the last row of the result set.

```
cout << "LAST" << endl;
row = cursor.NextRow(0, ITPositionLast);
if (row != NULL)
{
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

g. Fetch the 500th row from the result set.

```
cout << "500th" << endl;
row = cursor.NextRow(0, ITPositionAbsolute, 500);
if (row != NULL)
{
    rowcount++;
    cout << row->Printable() << endl;
    row->Release();
}
```

The cursor model in the Object Interface for C++ adheres to the following rules:

- When the cursor is first opened, it is moved before the first row. When you retrieve a row, the cursor advances to the row and then retrieves the data.
- When a cursor reaches the last row in a set it has scrolled through and a subsequent fetch returns `NULL`, the cursor remains moved on the last row. If you reverse the direction of the subsequent fetch to retrieve the previous row, then the second-to-last row is fetched.

- If you fetch from the last row up to the first row until there are no more rows, the cursor remains moved on the first row.
- Cursors do not wrap around. For example, you cannot open a cursor and retrieve the previous row in an attempt to wrap around to the last row. Similarly, you cannot wrap around from the last row to the first row.
- When using **ITPositionAbsolute** to move the cursor, use 1 for the first row.

7. The application can modify the columns of the fetched row by using, for example, FromPrintable().

```
if(!colduprow->FromPrintable("2"))
    {
    cerr << "Couldn't set the column value" << endl;
    return -1;
    }
else
    {
    cout << "Column 0 is now " << colduprow->Printable() << endl;
    }
```

8. If the cursor was opened as updateable, the application can update the current row by using the UpdateCurrent() function, or delete it using DeleteCurrent(). The application can use the IsUpdatable() function to check whether the cursor can be updated. Calling UpdateCurrent() causes modifications that have been made to the current row to be reflected in the database. The current row being the row that was most recently returned by the Fetch() or the NextRow() function.

```
if(!cursor.UpdateCurrent())
{
    cerr << "Could not update the current row" << endl;
    return -1;
}
```

If the application fetches the row, holds its reference, and then fetches another row, the first row is no longer current, and updates to it are not reflected in the database when the application calls UpdateCurrent().

The application can close the cursor, modify parameters, and reopen the cursor. Reopening a cursor closes the current one. Parameter values that have not been reset stay the same.

After the application finishes with the cursor, it drops the cursor by using the Drop() function. The same instance of **ITCursor** can be used to prepare another cursor by calling Prepare(), which calls Drop() for the current cursor.

Using the large object manager

The **ITLargeObjectManager** class performs simple operations on large objects such as creating, opening, reading, and seeking.

The functionality of the **ITLargeObjectManager** class is only supported with HCL Informix® databases.

Generally, this class is not used directly, but is included as a member of some class that implements a database type that has one or more large objects within it. For instance, a server sound data type might have a large object that holds the digitized waveform. The C++ type implementation must know how to read that large object. By using an

ITLargeObjectManager as a member, the implementor of the data type can use code from the **ITLargeObjectManager** class implementation.

The application can use ITLargeObjectManager::CreateLO() to create a large object. It can then get the handle of the newly created large object in either text or binary form by using ITLargeObjectManager::HandleText() or ITLargeObjectManager::Handle() and insert it into a table. These operations must occur within the same transaction; otherwise the large object falls prey to garbage collection.

You can perform operations on large objects within a fetched row even though the connection is still checked out (locked). A connection is checked out after the ITQuery::ExecForIteration() method returns multiple rows in the result set. It remains checked out until either the last row in the result set has been fetched with ITQuery::NextRow() or the query processing has been terminated by calling ITQuery::Finish(). While a connection is checked out, no other query can be executed on that connection.

The following excerpt from `loadtab.cpp` illustrates the use of the **ITLargeObjectManager**.

To use the **ITLargeObjectManager**, the application creates an instance of it on an opened connection object. The CreateLO() method creates the large object and sets the handle of the **ITLargeObjectManager** to the new large object.

The Write() method writes the string pointed to by `pdb` into the large object from the current position (in this case from the beginning of the string).

Finally, the statement parameter is set to the value of the large object handle, retrieved in text format by calling **ITLargeObj**.

```
ITLargeObjectManager lobMgr(conn);
    lobMgr.CreateLO();
    lobMgr.Write(pdb, strlen(pdb));

    if (!param->FromPrintable(lobMgr.HandleText()))
        {
        cerr
            << "Could not set LOB parameter "
            << paramno << " to '" << pdb << "'" << endl;
        return -1;
        }
    }
else if(param->TypeOf().Name().Equal("byte"))
    {
    ITDatum *pdatum = 0;
    param->QueryInterface(ITDatumIID, (void **)&pdatum);
    if(!pdatum)
        {
        cerr << "BYTE type does not expose ITDatum???" << endl;
        return -1;
        }
    if(!pdatum->SetData(pdb, pdbpos, 0))
        {
        cerr << "SetData() for BYTE failed" << endl;
        return -1;
        }
    pdatum->Release();
    }
else if (null == TRUE)
    {
    if (!param->SetNull())
```

```
        {
        cerr << "Could not set parameter "
             << paramno << " to null" << endl;
        return -1;
        }
    }
```

Using ITRoutineManager

The **ITRoutineManager** class provides an alternative way to execute server routines. The functionality of the **ITRoutineManager** class is only supported with HCL Informix® databases.

When using **ITRoutineManager**, a connection does not have to be checked out to get or execute a routine (and a value object, therefore, can use it), and the execution of the routine commences faster since there is no SQL to parse.

The following excerpts from `routine.cpp` illustrate the use of **ITRoutineManager**.

1. To use **ITRoutineManager**, the application creates an instance of it on an open connection object.

   ```
   ITRoutineManager routine(conn);
   ```
2. The GetRoutine() method retrieves the function descriptor for the function whose signature is passed as an argument.

   ```
   ITBool bret = routine.GetRoutine("function sum(int,int)");
   ```
3. The application sets parameter values by using ITValue::FromPrintable().

   ```
   val = routine.Param(0);
   val->FromPrintable("1");
   val->Release();
   ```

   It can also set parameter values by using ITRoutineManager::SetParam().
4. The routine is executed with ExecForValue(), which returns a pointer to **ITValue** corresponding to the return value of the routine.

   ```
   val2 = routine.ExecForValue();
   ```
5. A Release() call releases the **ITValue** instance.

   ```
   val2->Release();
   }
   ```

Access data values

This section describes the specific value interfaces in detail, and shows how to modify value objects and extract information through the value interfaces into host variables in your application.

Access data values

A column value in a database can be an atomic SQL92 type (such as **integer** or **varchar**) or, in HCL Informix® databases, any of the following extended data types:

- An opaque data type, such as those supplied with HCL Informix® DataBlade® modules and extensions (for example **binaryvar** for binary data)
- Row types, including types that use inheritance
- Collection types, such as **Set**, **List**, and **Multiset**
- Large object types

To enable applications to interact uniformly with value objects, all value objects present the **ITValue** interface. Value objects can expose additional interfaces to present different behaviors to the application. For instance, a value object representing a set can expose a container interface such as **ITSet** or **ITContainer**.

The following table lists the Informix® value object interfaces.

Interface	Description
ITRow	Row object interface (for example, a vector of named attributes, such as a row)
ITContainCvt	Container object with members that can be converted to and from C++ types
ITContainer	Container object with integer index-based access
ITConversions	Object that can be converted to and from C++ base types
ITDateTime	Date and time information
ITDatum	Underlying data access
ITErrorInfo	Error information
ITEssential	Base interface. Supports reference counting and interface querying
ITLargeObject	Large object. Supports file read/write semantics
ITSet	Container object with random access
ITValue	Basic value object interface

For a table showing how the server data types are supported in the , see Supported data types on page 89.

Value object management

All value object interfaces are derived from the base interface, **ITEssential**. This interface defines basic reference counting methods (AddRef and Release) on objects. Reference counting enables applications to ensure that the references to objects remain valid.

The ITEssential::QueryInterface method enables an application to determine whether an object supports a specified interface, either one defined by the or a custom interface created by a DataBlade® developer. If the interface is supported, ITEssential::QueryInterface provides a pointer to the interface and returns IT_QUERYINTERFACE_SUCCESS. If the interface is not supported, ITEssential::QueryInterface returns IT_QUERYINTERFACE_FAILED. For a list of interface identifiers for the interfaces provided by the Object Interface for C++, see The ITEssential interface on page 84.

Because all value object interfaces derive from **ITEssential**, your application can obtain a pointer to any interface supported by the value object from any other interface supported by the object.

The `tabcnt.cpp` example reads an integer value (the number of tables in the database) from the server into a value object, then converts it into a host variable by using the **ITConversions** interface. The following code excerpts illustrate the use of the QueryInterface method in the `tabcnt.cpp` example:

1. Issue the query that returns the number of tables.

```
ITRow *row;
row = q.ExecOneRow("select unique count(*) from systables
where tabname in ('systables', 'syscolumns',
'sysviews');");
```

2. Extract the value object from the first column of the result row.

```
ITValue *v = row->Column(0);
```

3. Extract an **ITConversions** interface from the object.

```
ITConversions *c;

// Extract an interface. The return code IT_QUERYINTERFACE_SUCCESS
// should be used for compatibility reasons.
if (v->QueryInterface(ITConversionsIID, (void **) &c)
    == IT_QUERYINTERFACE_SUCCESS)
{
```

4. Convert the value into a host variable, print the value, and release the conversions interface.

```
int numtabs;
if (c->ConvertTo(numtabs))
{
    cout << "Number of rows in the query was: " << numtabs << endl;
}

// Release the conversions interface
c->Release();
```

5. Release the **ITValue** and **ITRow** interfaces.

```
v->Release();
row->Release();
```

Objects are created with a reference count of 1 when they are returned to the application. When your application calls ITEssential::QueryInterface and obtains a pointer to an interface, another reference to the object is returned to the application, and the reference count is incremented. When the application no longer requires an interface, it must call the **Release** method to release the interface.

The ITValue interface

The **ITValue** interface defines simple comparison and printing methods on a value object and provides access to the server type information of an object.

All value objects must, at a minimum, expose an **ITValue** interface or an interface derived from **ITValue**. An object can expose other interfaces accessible through the ITEssential::QueryInterface method.

The ITValue::TypeOf method returns a reference to an **ITTypeInfo** object, from which your application can extract information such as its server type, whether it is a simple or collection type, its size (fixed or variable), and other information. For more details, see The ITTypeInfo class on page 76.

Other **ITValue** methods enable your application to perform comparisons to determine whether the object is equal to, greater than, or less than another object. To determine whether objects are comparable, your application can call the ITValue::CompatibleType method. The ITValue::CompatibleType method is defined by the implementor of a value object. The ITValue::CompatibleType method more loosely defines comparisons than the ITValue::SameType method, enabling applications to compare value objects of different types.

Two types are said to be compatible if they meet any of the following conditions:

- They are the same type.
- They are built in types that expose **ITDateTime** (date, datetime, interval).
- They both expose the **ITConversions** interface.
- They are DISTINCT from the same type.
- They are row types with the same column types.
- They are collection types with the same constructor and member types.

For instance, all value objects implemented by HCL Informix® that expose an **ITDateTime** interface are defined to be compatible.

Value objects can be updated by using the FromPrintable() function or set to `NULL` using SetNull(). The application can determine whether the object was updated by calling the IsUpdated() function.

The ITConversions interface

The **ITConversions** interface is exposed by objects that can be converted to and from C++ host variable type instances.

The conversion methods are of the form **ITBool ITConversions::ConvertTo(*base_type*)**. The **cnvex.cpp** example attempts to determine whether the value object that has exposed an **ITConversions** interface through an interface pointer is convertible to **int**, **double**, and other types.

For details about converting the columns of a row to C++ built-in types, see The ITContainerIter class on page 51.

The application can use **ITConversions::ConvertFrom(*base_type*)** to set the value object to a C++ base type value.

The ITDatum interface

The **ITDatum** interface is derived from **ITValue** and provides additional methods to get and set the underlying binary data and to obtain the connection object on which the value object was created. Value objects expose **ITDatum** to be able to participate in complex object updates.

The ITDatum::Data() method returns the (constant) pointer to the binary data. The memory for this data is managed by the object. An application does not attempt to modify the memory returned by Data(). For text data, Data() returns the pointer to MI_LVARCHAR, for row data, the pointer to MI_ROW, and for collections, the pointer to MI_COLLECTION.

The ITDatum::DataLength() method returns the length of underlying data. For opaque structures (such as MI_ROW and MI_COLLECTION), the value returned by DataLength() is not meaningful to the application.

The ITDatum::SetData() method sets the value object data to the data provided as the argument. The data must be in the same form as returned by ITDatum::Data(). For opaque structures the data length is ignored.

The ITDatum::Connection() method returns (by reference) the connection object that was used in the instantiation of the value object.

Generally, the C++ Interface uses ITDatum() members to update the row or collection of objects.

The ITDateTime interface

The **ITDateTime** interface can be exposed by value objects that represent a time-based value.

The following example shows how an application uses a pointer to an **ITDateTime** interface to extract time-based information and print it.

```
ITDateTime *dt;

// Extract an interface. The return code IT_QUERYINTERFACE_SUCCESS
// should be used for compatibility reasons.
if (v->QueryInterface(ITDateTimeIID, (void **) &dt)
    == IT_QUERYINTERFACE_SUCCESS)
{
    cout << "The date value is: " << endl
         << "Year:" << dt->Year() << endl
         << "Month: " << dt->Month() << endl
         << "Day: " << dt->Day() << endl
         << "Hour: " << dt->Hour() << endl
         << "Minute: " << dt->Minute() << endl
         << "Second: " << dt->Second() << endl;

    // Release the Date/Time interface
    dt->Release();
}
```

The application can use the ITDateTime::FromDate and ITDateTime::FromTime methods to set the date and time portions of a datetime object. If an object contains both date and time information and, for example, **FromDate** is called, the value of the time portion of an object does not change.

The ITLargeObject interface

The **ITLargeObject** interface is exposed by value objects that must expose to their underlying data a functionality similar to that of a file I/O interface. Typically, such objects represent server data types that are derived from or contain a server smart large object type instance.

This functionality of this interface is supported only with HCL Informix® databases.

The following excerpt illustrates how the large object interface is extracted:

```
ITLargeObject *loif;
if (v->QueryInterface(ITLargeObjectIID, (void **) &loif)
    == IT_QUERYINTERFACE_SUCCESS)
```

The following loop reads data from the large objects and writes it to **cout**:

```
while ((n = loif->Read(buf, sizeof(buf))) > 0)
{
    cout.write(buf, n);
}
cout.flush();
```

The ITErrorInfo interface

The **ITErrorInfo** interface includes methods that manage errors from the server or from the library. The **ITErrorInfo** interface enables your application to set callback routines that are called when an error occurs.

For details, see The ITErrorManager class on page 57.

This functionality of this interface is only supported with Informix® databases.

Value objects such as large objects and set interface objects that have methods that cause interactions with the server expose the **ITErrorInfo** interface. The following excerpts illustrate the correct use of the **ITErrorInfo** interface:

1. Extract the large object interface.

   ```
   ITErrorInfo *errif;
   if (v->QueryInterface(ITLargeObjectIID, (void **) &loif)
       == IT_QUERYINTERFACE_SUCCESS)
   {
   }
   ```

2. Extract the error management interface.

   ```
   // Extract the errorinfo interface.
   if (v->QueryInterface(ITErrorInfoIID, (void **) &errif)
       == IT_QUERYINTERFACE_SUCCESS)
   {
   }
   ```

3. Close the connection before reading the large object.

   ```
   conn.Close();
   ```

   This induces an error.

4. Check byte count. If 0 bytes were read, check to see if an error occurred.

   ```
   if (size == 0)
   {
       // No bytes were read. Was there an error?
       if (errif->Error())
       {
           cerr << "Zero bytes read. Server error was" << endl
                << errif->ErrorText() << endl;
       }
   }
   ```

The ITRow interface

The **ITRow** interface is derived from **ITValue** and is the primary interface for interacting with objects that represent database rows.

For details, see The ITRow interface on page 86.

The ITSet interface

The **ITSet** interface can be exposed by an object that contains other objects and can provide arbitrary or nonsequential access to the underlying objects.

The ITQuery::ExecToSet method provides random access to the result of a **select** query by returning this object. For an example of an object that exposes the **ITSet** interface, see the example file **rowset.cpp**.

Container objects that expose the **ITSet** interface are especially useful in GUI applications, because the random-access capabilities of the **ITSet** interface can be used in association with a scroll bar to support scrolling through the result set.

The following code excerpts from the **rowset.cpp** example illustrate the basic object container features of the row set object created by a call to ITQuery::ExecToSet:

1. Execute a **select** statement and return a value object that exposes an **ITSet** interface.

   ```
   ITSet *set = q.ExecToSet(qtext);
   if (set == NULL)
       {
   ```

2. Open the set.

   ```
   if (!set->Open())
       {
       }
   ```

3. Fetch value objects from the set.

   ```
   while ((value = set->Fetch()) != NULL)
       {
   ```

   In a graphical user interface (GUI) program, the application might move to a location within the set that corresponds to the setting of a scroll bar before fetching data.

4. Perform tasks with the value objects, releasing any interfaces when finished.

   ```
   if (value->QueryInterface(ITRowIID, (void **) &row)
       == IT_QUERYINTERFACE_FAILED)
       {
       cout << "Could not get row interface..." << endl;
       }
   else
       {
       cout << row->Printable() << endl;
       row->Release();
       }
   ```

```
rowcount++;
value->Release();
```

5. Close the set.

```
if (!set->Close())
    {
    }
```

6. Release the set.

```
set->Release();
```

The application can use the ITSet::Insert method to insert new members into the container objects and TSet::Delete() to remove a member.

The ITContainer interface

The **ITContainer** interface is exposed by a value object that contains other objects and does not support a concept of current position within the set. Instead, the **ITContainer** interface uses an index to extract the corresponding object.

The **ITContainer** object can be exposed to enable applications to use the **ITContainerIter** class to iterate over the result set and extract values into C++ base type host variables.

The example program `fsexamp1.cpp` builds a temporary **ITContainerIter** object to iterate over the result row of a query returned by ITQuery::ExecOneRow. The **ITContainerIter** object constructor implicitly extracts an **ITContainer** interface from the object it is constructed against, or an **ITContainCvt** interface if possible. The approach illustrated by the `fsexamp1.cpp` example is more efficient than that used by the `tabcnt.cpp` example (which performs similar processing).

The following code excerpts point out relevant passages from the `fsexamp1.cpp` example.

1. Build the query object.

```
ITQuery q(conn);
```

2. Issue the query.

```
ITRow *row =
    q.ExecOneRow("select unique count(*) from systables where tabname
        in ('systables', 'syscolumns', 'sysviews');");

if (q.Error())
    {
    // some error processing row
    cerr << q.ErrorText() << endl;
    return 1;
    }
```

3. Build an **ITContainerIter** object on the result row, and extract a C++ **int** value.

```
int numtabs;
ITContainerIter(row) >> numtabs;
```

4. Release the underlying row.

```
row->Release();
```

The ITContainCvt interface

The **ITContainCvt** interface combines the features of the **ITContainer** and **ITConversions** interfaces.

The **ITContainCvt** interface can be exposed by objects that are containers of base type instances, such as data types that include an array of values like a polygon or path. Unlike the **ITContainer** interface, the constituent values are converted by the container object directly into C++ host types, instead of into other value objects.

The `contain.cpp` example uses a sample array value object, and extracts an **ITContainCvt** interface from the array object to load values from the array into application variables. (The `contain.cpp` example uses a distinct data type, and so it is only supported with Informix®.) The following excerpts point out use of the **ITContainCvt** interface:

1. Execute a query that returns an array.

```
ITRow *row =
    q.ExecOneRow("select * from bitarraytab;");
```

2. Extract the array value from the result row.

```
ITValue *arrayval = row->Column(0);
```

3. Extract an **ITContainCvt** interface from the object and release the interfaces that are no longer required.

```
ITContainCvt *arraycont;
arrayval->QueryInterface(ITContainCvtIID, (void **) &arraycont);
row->Release();
arrayval->Release();
```

4. Iterate over the **ITContainCvt** interface and extract the array values into application variables.

```
// The iterator class iterates over every member
// of an object
// exposing an ITContainer or ITContainerCvt interface.
ITContainerIter iter(arraycont);

// Add all the items to the stream
char buf[8192];
ostrstream  cstream(buf, sizeof buf);

for (int i = 0; i < arraycont->NumItems(); i++)
{
   int value;
    iter >> value;
    cstream << '[' << i << ']' << " = " << value << endl;
}
```

5. Release the **ITContainCvt** interface.

```
arraycont->Release();
```

Create and extend value objects

When you retrieve values from the HCL Informix® database by using the , the values are returned as value objects. Value objects are created by the Object Interface for C++ by using an extensible class factory that maps Informix® data types to C ++ objects.

Important: Only Informix® supports extensible data types. Therefore, the information in this section applies only to Object Interface for C++ applications that connect with Informix® databases.

The value object approach enables DataBlade® developers to create objects that represent new server data types and ensures that client applications can operate with these new data types. Client applications that use the value object approach do not depend on the representation of the object in the database and continue to run if the database representation and the corresponding value object implementation changes. For details about library support for value objects, see Dynamic loading on page 47.

The raw data object

If the class factory for a specific server type is not registered, the automatically creates an object that exposes both an **ITValue** interface and an **ITDatum** interface. To obtain a pointer to the binary data of the object, use the ITDatum::Data method. The resulting pointer can be used to access the data structure that corresponds to the object.

This approach violates the principle of information hiding. By accessing the structure through a pointer, the user of the object creates a dependency on the particular implementation of an object. If that implementation changes, the applications that use the object can cease to function. The interface approach to object encapsulation ensures that an application cannot create a dependency on a particular implementation of an object.

The `rawval.cpp` example shows how an application can use the **ITDatum** interface to extract a data structure from the value object returned from the Object Interface for C++ when no specific value object constructor is found for the server type. This example application retrieves a pointer to a sequence of bytes from the server. The following code excerpts point out use of the raw data interface.

1. Issue a query to return an array and extract the value from the row.

```
ITQuery q(conn);

ITRow *row =
    q.ExecOneRow("select byte_val from regresstab;");
// Extract the column
ITValue *v;
v = row->Column(0);
```

2. Extract the **ITDatum** interface from the object.

```
ITDatum *rv;
if (v->QueryInterface(ITDatumIID, (void **) &rv) ==
    IT_QUERYINTERFACE_SUCCESS)
    {
```

3. Extract the data pointer from the object into an application pointer.

```
char *pch = (char *)rv->Data();
```

4. Search the data types for a match.

```
char match[] = "Informix";

char *found = strstr(pch, match);
```

5. Release the **ITDatum** interface.

```
rv->Release();
```

Build simple value objects

Most DataBlade® developers want to create true value objects for new types. The simplest way to do so is to derive a C++ class directly from the **ITDatum** interface class. You must then add to the new class:

- Implementation for all the **ITDatum** methods, all of which are pure virtual.
- Any data members needed to hold the data of the object
- A few data members required to support the **ITDatum** methods; in particular, an **ITTypeInfo** object.
- A class constructor and destructor.

If your value object code is to be directly linked with your application, you must add:

- A static class factory function that calls the class constructor.

  The class factory function must accept an instance of an **ITMVDesc** structure.
- A global **ITFactoryList** object that registers the class factory function under a server type name.

If you want to use dynamic loading feature, you must provide:

- A C-linkage factory function that calls the class constructor.

  The function must accept an instance of an **ITMVDesc** structure.
- An entry in the map file for this class.

  For details, see Dynamic loading on page 47.

The `simpval.cpp` example illustrates the use of the **ITMVDesc** descriptor and **ITDatum** interface. The `simpval.cpp` example creates a true value object for the **bitarray** data type.

The following `simpval.cpp` code excerpts show how to create a true value object:

1. Define the data structures for holding the bit array objects.

```
typedef mi_integer bitarray_t;
```

2. Define the array of integers class from **ITDatum**, implementing methods for the **ITDatum** abstract methods.

```
class Bitarray : public ITDatum
{
public:

    // Overrides of ITEssential methods
    virtual ITOpErrorCode IT_STDCALL QueryInterface
                                        (const ITInterfaceID &ifiid,
                                                  void **resultif);
    virtual unsigned long IT_STDCALL AddRef();
    virtual unsigned long IT_STDCALL Release();
```

```
    // Overrides of ITValue methods
    virtual const ITString & IT_STDCALL Printable();
    virtual const ITTypeInfo & IT_STDCALL TypeOf();
    virtual ITBool IT_STDCALL IsNull();
    virtual ITBool IT_STDCALL SameType(ITValue *);
    virtual ITBool IT_STDCALL CompatibleType(ITValue *);
    virtual ITBool IT_STDCALL Equal(ITValue *);
    virtual ITBool IT_STDCALL LessThan(ITValue *);
    virtual ITBool IT_STDCALL IsUpdated();
    virtual ITBool IT_STDCALL FromPrintable(const ITString &);
    virtual ITBool IT_STDCALL SetNull();

    // Overrides of ITDatum methods
    virtual MI_DATUM IT_STDCALL Data();
    virtual long IT_STDCALL DataLength();
    virtual ITBool IT_STDCALL SetData(MI_DATUM, long, ITPreserveData *);
    virtual const ITConnection & IT_STDCALL Connection();

    // Class constructor, destructor
    Bitarray(ITMVDesc *);
    ~Bitarray();

    // Factory Constructor -- this is the entry point for objects to
    // be created. It uses the class constructor to build an object
    // and returns in to the caller. It is called automatically by the
    // Interface when an object of the "bitarray" type is returned by
    // the server to the interface
    static ITValue *MakeValue(ITMVDesc *);

    // Data members to implement ITEssential functionality
    long refcount;

    // Data members to implement ITValue functionality
    ITTypeInfo typeinfo;
    ITBool isnull, isupdated;
    ITString printable_value;

    // Data members to implement bitarray storage
    bitarray_t value;

    ITConnection conn;
};
```

3. Construct the object, initializing its reference count and data and type information.

```
Bitarray::Bitarray(ITMVDesc *mv)
    : refcount(1),
      typeinfo(*mv->vf_origtypeinfo),
      isupdated(FALSE),
      conn(*mv->vf_connection)
{
    // NULL?
    isnull = mv->vf_libmivaluetype == MI_NULL_VALUE;
    if(!isnull)
        value = *(bitarray_t *)mv->vf_data;
}
```

4. Define the factory entry point for the object.

```
ITFactoryList BitarrayFactory("bitarray",
                              Bitarray::MakeValue);
```

When this object file is linked into the application, the linker forces the construction of the **BitarrayFactory** variable to take place before the application begins to execute. The **ITFactoryList** constructor puts the mapping from server type to **Bitarray::MakeValue** into the global type mapping list.

5. Implement the factory entry point, which must be a static member function instead of a method, because at the time the factory entry point is called there is no object on which to call a method.

```
ITValue *
Bitarray::MakeValue(ITMVDesc *mv)
{
    return new Bitarray(mv);
}
```

This function builds a new **Bitarray** object and returns it. Because the object derives from the **ITDatum** interface, it is valid to return the object itself instead of calling ITEssential::QueryInterface on the object to extract the correct interface.

6. Define the ITEssential::QueryInterface function and the reference count methods.

```
ITOpErrorCode
Bitarray::QueryInterface(const ITInterfaceID &iid,
                                void **ifptr)
{
    int result = IT_QUERYINTERFACE_SUCCESS;

    switch (ITIIDtoSID(iid))
    {
        case ITEssentialSID:
        case ITValueSID:
        case ITDatumSID:
            *ifptr = this;
            break;

        default:
            result = IT_QUERYINTERFACE_FAILED;
            *ifptr = NULL;
            break;
    }
    if (result == IT_QUERYINTERFACE_SUCCESS)
        AddRef();

    return result;
}
```

7. Implement the **ITDatum** methods appropriate for the object.

```
const ITString &
Bitarray::Printable()
{
    if(IsNull())
        return printable_value = "null";

    char buf[32];
    ostrstream cstream(buf, sizeof buf);
    cstream << value << ends;
```

```
    return printable_value = cstream.str();
}
```

Expose multiple interfaces

If an object must expose multiple behaviors, the object must be able to return multiple interfaces. To enable an object to return multiple interfaces, you can derive the object from the various interfaces by using multiple inheritance, or derive the object from a separate implementation hierarchy and derive nested classes from the appropriate interfaces.

The nested class solution, which is used by the , has the following benefits:

- It allows the COM-compliant exposure of multiple interfaces.
- It allows delegation, the ability of a container class to expose an interface belonging to a class it contains. For more details, see Object Containment and Delegation on page 44.
- It creates multiple implementations of reference counting code for each interface, making it easier to track the reference counts for each interface. By tracking references to individual interfaces, your application can optimize object storage by allocating or deallocating part of an object based on whether a specific interface has an outstanding reference count. For example, if an object exposes **ITLargeObject** and it uses **ITLargeObjectManager** to implement its functions, it can call ITLargeObjectManager::Close() when the **ITLargeObject** interface reference count drops to 0 so that the number of open smart large objects is minimized.

For a demonstration of the nested-class model, see the `ifval.cpp` example. The `ifval.cpp` example is driven by the `contain.cpp` example application.

The following code excerpts from `ifval.cpp` illustrate the implementation of an array of integers value object that exposes both **ITDatum** and **ITContainCvt** interfaces:

1. Define the private data structures.

   ```
   typedef mi_integer bitarray_t;
   ```

   This structure is not exposed to the application.
2. Define the object class. Instead of using inheritance on the parent object, use nested classes to define the individual interfaces.

   ```
   class Bitarray
   {
   public:
       // ITDatum-derived nested class. This just passes work through
       // the parent pointer into the parent object
       class XITDatum : public ITDatum
       {
       public:
           // ...
       } datum_interface;

       // ITContainCvt-derived nested class
       // This just passes work through the parent
       // pointer into the parent object
   ```

```
    class XITContainCvt : public ITContainCvt
    {
    public:
        // ...
    } containcvt_interface;
    // ...
};
```

3. Build the object.

```
// Implementation
Bitarray::Bitarray(ITMVDesc *mv)
    : refcount(1),
      typeinfo(*mv->vf_origtypeinfo),
      conn(*mv->vf_connection),
      isupdated(FALSE)
{
  // NULL?
  isnull = mv->vf_libmivaluetype == MI_NULL_VALUE;

  // set up interfaces
  datum_interface.parent = this;
  containcvt_interface.parent = this;

  if(!isnull)
    value = *(bitarray_t *)mv->vf_data;
}
```

4. Define the class factory mapping and entry point.

```
ITFactoryList BitarrayFactory("bitarray",
                                    Bitarray::MakeValue);

// Create the Bitarray object, and return pointer to
// it's ITValue implementation
ITValue *
Bitarray::MakeValue(ITMVDesc *mv)
{
    Bitarray *impl = new Bitarray(mv);
    return (ITValue *)&impl->datum_interface;
}
```

5. Define the base class methods for objects and return the address of the nested interfaces when requested by the application.

```
ITOpErrorCode
Bitarray::QueryInterface(const ITInterfaceID &iid,
                                 void **ifptr)
{
    int result = IT_QUERYINTERFACE_SUCCESS;

    // Return different interfaces as appropriate by referencing
    // nested class members.
    switch (ITIIDtoSID(iid))
    {
        case ITEssentialSID:
        case ITValueSID:
        case ITDatumSID:
            *ifptr = (void *) &datum_interface;
            break;
```

```
        case ITContainCvtSID:
            *ifptr = (void *) &containcvt_interface;
            break;

        default:
            result = IT_QUERYINTERFACE_FAILED;
            *ifptr = NULL;
            break;
    }
    if (result == IT_QUERYINTERFACE_SUCCESS)
        AddRef();

    return result;
}
```

This object does not support delegation, so there is only one real **QueryInterface** implementation on the object.

6. Define the reference counting code.

```
unsigned long
Bitarray::AddRef()
{
    return ++refcount;
}

unsigned long
Bitarray::Release()
{
    if (--refcount <= 0)
    {
        delete this;
        return 0;
    }
    else
    {
        return refcount;
    }
}
```

7. Implement the **ITDatum** interface methods.

```
const ITString &
Bitarray::Printable()
{
    if(IsNull())
        return printable_value = "null";
    char buf[32];
    ostrstream cstream(buf, sizeof buf);
    cstream << value << ends;
    return printable_value = cstream.str();
}
```

8. Implement the **ITContainCvt** interface.

```
ITBool
Bitarray::ConvertTo(long item, int &dbvalue)
{
    if (IsNull() || item >= NumItems())
        return false;
    dbvalue = !!(value & (1 << (NBITS - 1 - item)));
```

```
    return true;
}
```

This interface converts the member value from the object into a host variable.

9. Declare pass-through methods for the nested interfaces. The methods call the corresponding method in the parent class.

```
ITOpErrorCode
Bitarray::XITDatum::QueryInterface(const ITInterfaceID &ifiid,
                                   void **resultif)
{
    return parent->QueryInterface(ifiid, resultif);
}

unsigned long
Bitarray::XITDatum::AddRef()
{
    return parent->AddRef();
}

unsigned long
Bitarray::XITDatum::Release()
{
    return parent->Release();
}

const ITString &
Bitarray::XITDatum::Printable()
{
    return parent->Printable();
}

const ITTypeInfo &
Bitarray::XITDatum::TypeOf()
{
    return parent->TypeOf();
}

ITBool
Bitarray::XITDatum::IsNull()
{
    return parent->IsNull();
}

ITBool
Bitarray::XITDatum::SameType(ITValue *v)
{
    return parent->SameType(v);
}

ITBool
Bitarray::XITDatum::CompatibleType(ITValue *v)
{
    return parent->CompatibleType(v);
}

ITBool
```

```
Bitarray::XITDatum::Equal(ITValue *v)
{
    return parent->Equal(v);
}

ITBool
Bitarray::XITDatum::LessThan(ITValue *v)
{
    return parent->LessThan(v);
}

ITBool
Bitarray::XITDatum::IsUpdated()
{
    return parent->IsUpdated();
}

ITBool
Bitarray::XITDatum::FromPrintable(const ITString &v)
{
    return parent->FromPrintable(v);
}

ITBool
Bitarray::XITDatum::SetNull()
{
    return parent->SetNull();
}

ITOpErrorCode
Bitarray::XITContainCvt::QueryInterface(const ITInterfaceID &ifiid,
                                        void **resultif)
{
    return parent->QueryInterface(ifiid, resultif);
}

unsigned long
Bitarray::XITContainCvt::AddRef()
{
    return parent->AddRef();
}

unsigned long
Bitarray::XITContainCvt::Release()
{
    return parent->Release();
}

ITBool
Bitarray::XITContainCvt::ConvertTo(long item, int &value)
{
    return parent->ConvertTo(item, value);
}

long
Bitarray::XITContainCvt::NumItems()
{
```

```
    return parent->NumItems();
}

ITBool
Bitarray::XITContainCvt::ConvertFrom(long item, int value)
{
    return parent->ConvertFrom(item, value);
}
```

Value objects and connection events

Value objects are created in the following circumstances:

- **Query** objects create instances of the top-level rows.
- Complex objects (rows and collections) create instances of their members.
- Prepared query objects create instances of their parameters.

If the value object encapsulates a small, fixed-size datum, it can keep a local copy of that datum. If the datum is large, of variable size, or represents a complex type, the value object keeps a pointer to it. To ensure that this pointer continues to be valid even after all the references to the object that owns the datum memory are released, **ITMVDesc** contains a pointer to the **ITPreserveData** interface of that object (**ITMVDesc.vf_preservedata**). The value object keeps this pointer and use it to call the AddRef() function when it is created. When the value object is deleted, it calls the Release() function by using the **ITPreserveData** pointer.

If the value object that keeps a local copy of a datum is updated, it modifies that local copy. If the value object keeps a pointer to the datum, it cannot modify that datum—it must create an instance of that datum and call the Release() function on the **ITPreserveData** pointer passed to it in **ITMVDesc**. The value object ensures that the IsUpdated() function of its **ITValue** interface returns `TRUE` if it is modified. The instance of a datum allocated on update is removed when the value object is removed. The `rowref.cpp` example illustrates this "allocate-on-update" technique.

To monitor connection events, a value object that keeps a pointer to row data can maintain a *connection stamp*.
This connection stamp, of type **ITConnectionStamp**, is checked before the row data pointer is dereferenced. The **ITConnectionStamp::EqualConnInstance** method of the **ITConnectionStamp** class can be used to tell if the connection is the same instance as that called by another connection stamp.

Use of the connection stamp and **ITPreserveData** interface is demonstrated in the `rowref.cpp` example source file, which is included in the `contain2.cpp` example application. The following code excerpts illustrate how the `rowref.cpp` example preserves a reference on its underlying data instead of copying the data:

1. Add a member variable to hold the **ITPreserveData** interface pointer and connection stamp.

```
class Bitarray
{
    ITPreserveData *preservedata;   // reference counter on datum
    ITConnectionStamp stamp;    // connection stamp
};
```

2. Initialize the **preservedata** member with the value from the descriptor, add a reference to it, and make a copy of the connection stamp.

```
Bitarray::Bitarray(ITMVDesc *mv)
    : refcount(1),
      typeinfo(*mv->vf_origtypeinfo),
      conn(*mv->vf_connection),
      stamp(mv->vf_connection->GetStamp()),
      preservedata(mv->vf_preservedata),
      isupdated(FALSE),
      pvalue(0)
{
    // NULL?
    isnull = mv->vf_libmivaluetype == MI_NULL_VALUE;

    // set up interfaces
    datum_interface.parent = this;
    containcvt_interface.parent = this;

    if(!isnull)
        {
        pvalue = (bitarray_t *)mv->vf_data;
        // We are holding an outstanding reference to datum, so
        // increment its owner's reference count.
        // Note that preservedata can be null for null objects.
        preservedata->AddRef();
}
```

3. When the object is being deleted, release the **preservedata** interface.

```
Bitarray::~Bitarray()
{
    if(isupdated)
        delete pvalue;
    else if(preservedata)
        preservedata->Release();
}
```

4. Before any attempt to de-reference the value member pointer, first check the connection stamp to ensure that the underlying data is still valid.

```
const ITString &
Bitarray::Printable()
{
    // If the underlying data has changed its not safe to proceed.
    if (!stamp.EqualConnInstance(conn.GetStamp()))
        return ITString::Null;

    if(IsNull())
      return printable_value = "null";

    char buf[32];
    ostrstream cstream(buf, sizeof buf);
    cstream << *pvalue << ends;
    return printable_value = cstream.str();
}
```

Create row type value objects

Object Interface for C++, Version 2.70 and later allows row or collection type value objects to be created by using the following methods.

Create row type value objects without an open connection

The process consists of two steps:

1. Create the ITTypeInfo object for the row type.
2. Instantiate the row type value object by using the ITFactoryList::DatumToValue() method and pass to it an ITMVDesc structure whose members are populated appropriately.

The row type object returned this way is a null row, which can be modified by using ITRow::FromPrintable(). Because the row type object has been created without an open connection, the underlying data of the row type value object cannot be modified with ITDatum::SetData() or retrieved with ITDatum::Data() (where **ITDatum** is an interface exposed by a row type value object). However, the remaining ITRow methods are not affected.

The following example illustrates how to create a row type value object without an open connection:

```
#include <iostream.h>
#include <it.h>
int
main()
{
             ITConnection conn;
             ITMVDesc desc;
             ITTypeInfo colType(conn,"integer", 4,-1,-1,-1,1);
             ITTypeInfo *ptrcolType = &colType;
             ITString colName = "int_val";
             ITTypeInfo newti(conn,"row(int_val integer)", 1,
                  &ptrcolType, &colName, NULL );
             desc.vf_origtypeinfo = (ITTypeInfo *) &newti;
             desc.vf_connection = &conn;
             desc.vf_typedesc = NULL;
             desc.vf_preservedata = NULL;
             desc.vf_outerunknown = NULL;
             desc.vf_datalength = newti.Size();
             desc.vf_libmivaluetype = MI_NULL_VALUE;
             desc.vf_data = NULL;
             ITValue *val = ITFactoryList::DatumToValue (desc);
             val->FromPrintable("row(1)");
             cout << val->Printable() << endl;
             val->Release();
}
```

Create collection type value objects without an open connection

You can create collection type value objects without an open connection by using a process similar to creating row types. As with row types, ITDatum::Data() and ITDatum::SetData() cannot be used to retrieve or modify values from a collection type created without an open connection.

The following example illustrates how to create a collection type value object without an open connection:

```
#include <iostream.h>
#include <it.h>
int
main()
{
          ITConnection conn;
          ITMVDesc desc;
          ITTypeInfo memberType(conn,"integer", 4,-1,-1,-1,1);
          ITTypeInfo newti( conn, "set(integer not null)",
          "set", memberType, NULL );
          desc.vf_origtypeinfo = (ITTypeInfo *) &newti;
          desc.vf_connection = &conn;
          desc.vf_typedesc = NULL;
          desc.vf_preservedata = NULL;
          desc.vf_outerunknown = NULL;
          desc.vf_datalength = newti.Size();
          desc.vf_libmivaluetype = MI_NULL_VALUE;
          desc.vf_data = NULL;
          ITValue *val = ITFactoryList::DatumToValue (desc);
          val->FromPrintable("set{1}");
          cout << val->Printable() << endl;
          val->Release();
}
```

Object Containment and Delegation

Objects that contain other objects are called container objects. There are two fundamental types of container objects:

Base type containers

Value objects that contain C++ base type instances (and do not contain other objects). For an example of base type containers, see Value objects and connection events on page 41.

Object containers

Value objects that contain other value objects. Object containers are created by using a technique called object delegation, in which the container object uses a predefined constituent object to define its subobjects.

Object delegation allows objects to be reused, as in C++ object inheritance, but protects against base-class fragility-the tendency for base classes to evolve beneath derived classes. Instead of deriving one class from another, the capabilities of one object are combined with the capabilities of another, through a process called interface delegation.

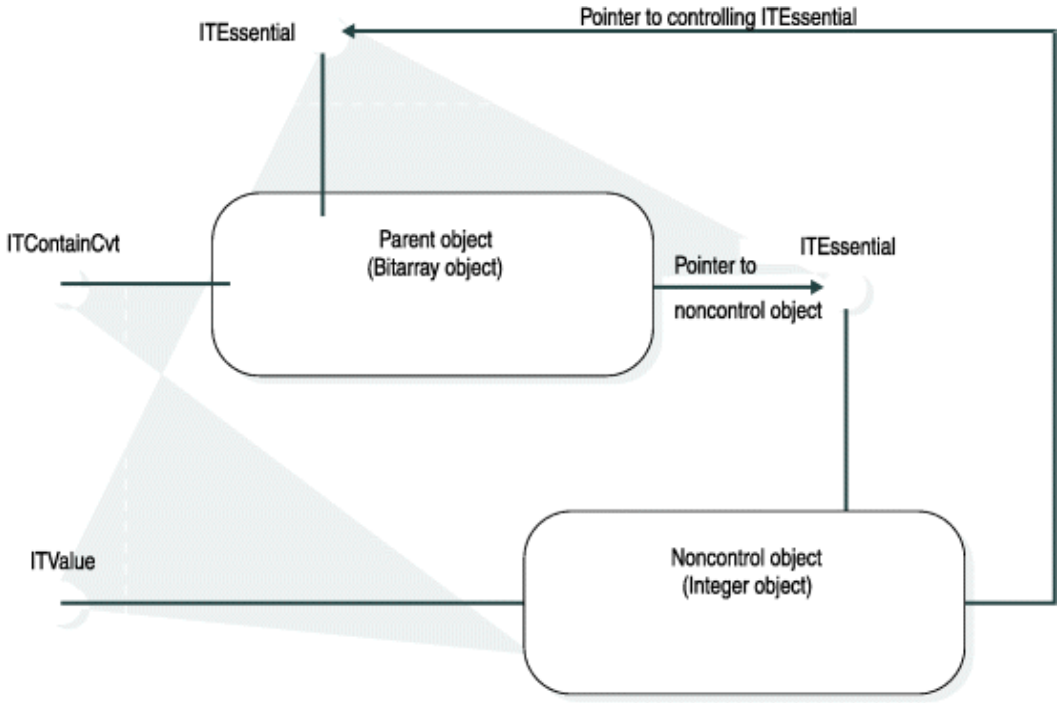
In interface delegation, a parent object exposes the interfaces of a contained object as if they were its own. The contained object is supplied with the controlling **ITEssential** pointer (in COM, a controlling unknown pointer) when it is constructed; this controlling **ITEssential** is the **ITEssential** interface of the parent object.

When any of the **ITEssential** methods of the delegated interface of the subobject are called (for example, QueryInterface, AddRef, and Release), they are delegated to the controlling **ITEssential** interface. For example, if the QueryInterface method of a delegated interface is called, the QueryInterface method of the parent object is called. Reference counting is also performed on the parent object rather than the subobject.

To ensure that the parent can extract interfaces from the subobject and delete it, the parent object must have a pointer to one interface that is not delegated. This interface is the **ITEssential** interface of the subobject, which must never be exposed outside of the parent object.

The following figure illustrates object delegation.

Figure 4. Object delegation



Object delegation is demonstrated by the `delegate.cpp` example, which is in turn driven by the `deldrv.cpp` example file. This example requires a bit array server data type and table defined by the following SQL statements:

```
create distinct type bitarray as integer;
create table bitarraytab (bitarraycol bitarray);
insert into bitarraytab values ('1');
```

The bit array value object implemented in the `delegate.cpp` example is created by aggregating the integer value object. Of the interfaces exposed by this subobject, only a few methods of the **ITContainCvt** interface of the container object and the **ITValue** interface of the integer value object are exposed outside of the bit array object. The interface of the integer value object is exposed through delegation.

A bit array is retrieved by the following query, which is issued in the `deldrv.cpp` example file:

```
select bitarraycol from bitarraytab;
```

The following excerpts from the `delegate.cpp` example show how to use object delegation to delegate the responsibility for creating objects to an **ITValue**-interface-exposing subobject within the **Bitarray** class:

1. Define the various **ITEssential** methods.

```
class Bitarray : public ITContainCvt
{
public:
    // Overrides of ITEssential methods
    virtual ITOpErrorCode IT_STDCALL QueryInterface(const ITInterfaceID &ifiid,
                                                    void **resultif);
    virtual unsigned long IT_STDCALL AddRef();
    virtual unsigned long IT_STDCALL Release();
```

2. Define the **ITContainCvt** methods. Because not all of the methods of the **ITContainCvt** interface of the nested object are used, the parent object cannot delegate the **ITContainCvt** interface to the subobject, as it does for the **ITValue** interface.

```
// Overrides of ITContainCvt methods
virtual long IT_STDCALL NumItems();
```

3. Define a pointer for the **ITEssential** interface of the subobject. The object must retain the **ITEssential** interface of the integer object, so it can release the subobject when the parent object is deleted. This interface is never passed back outside of a **Bitarray** object.

```
ITEssential *int_essential;
```

4. Define a pointer to hold an intermediate integer value object.

```
ITValue *int_value;
```

5. Make the **ITEssential** interface of **Bitarray** as the outer controlling unknown pointer.

```
desc.vf_outerunknown = this;
```

6. To create an integer subobject for delegation, the **Bitarray** constructor uses a local instance of **ITMVDesc**. This instance is identical to the **ITMVDesc** instance of **Bitarray**, except for the use of the integer **ITTypeInfo** that the **Bitarray** constructor retrieves by using ITTypeInfo::Source().

```
ITMVDesc desc = *mv;
desc.vf_origtypeinfo = (ITTypeInfo *)mv->vf_origtypeinfo->Source();
```

   The **ITMVDesc** instance is passed to ITFactoryList::DatumToValue() to instantiate the integer object and return a pointer to its **ITValue**. **Bitarray** retains this pointer for delegation.

7. Copy the **ITEssential** interface into a class member.

```
int_essential = desc.vf_outerunknown;
```

   The object constructor overwrites the **ITEssential** instance named **int_essential**.

8. When the object is deleted, release the interface of the integer subobject.

```
int_essential->Release();
```

9. If the application requests an interface that is not supported by this object, ask the integer subobject if it supports the interface.

```
ITOpErrorCode
Bitarray::QueryInterface(const ITInterfaceID &iid,
              void **ifptr)
{
    switch (ITIIDtoSID(iid))
        {
    case ITEssentialSID: case ITContainCvtSID:
        *ifptr = this;
```

```
        AddRef();
        return IT_QUERYINTERFACE_SUCCESS;

    default:
        // This object does not support the interface. Try the
        // delegated subobject...if the subobject supports the
        // interface, it will increment the reference counter on the
        // controlling unknown, so we don't need to increment it
        // here (except if you ask the subobject for its ITEssential
        // interface, in which case it will increment its own
        // reference count).
        return int_essential->QueryInterface(iid, ifptr);
        }
}
```

10. Implement the **ITContainCvt** methods.

```
// ContainCvt implementation
ITBool
Bitarray::ConvertTo(long item, int &dbvalue)
{
    if (int_value->IsNull() || item >= NumItems())
        return FALSE;
    const char *valasstr = int_value->Printable();
    int val = atoi(valasstr);
    dbvalue = !!(val & (1 << (NBITS - 1 - item)));
    return TRUE;
}

ITBool
Bitarray::ConvertFrom(long item, int val)
{
    if(NumItems() <= item)
        return FALSE;
    int value = val ? value | (1 << (NBITS - 1 - item))
        : value & ~(1 << (NBITS - 1 - item));
    char valasstr[32];
    sprintf(valasstr, "%d", value);
    return int_value->FromPrintable(valasstr);
}

long
Bitarray::NumItems()
{
  return NBITS;
}
```

Because of the way the **ITValue** interface is delegated, this forwarding is not necessary for the **ITValue** interface methods.

Dynamic loading

Dynamic loading is the feature that enables you to use shared object libraries to support value objects.

Using dynamic loading, if a client application receives from Informix® an object of a type for which it does not have a registered factory, the Object Interface for C++ factory system scans mapping files to determine whether there is a shared

object library that supports the type. If found, the library is loaded and the factory entry point is called to construct an object of the specified type for the client application.

Map files

The map file is a text file.

The format of the map file is:

```
[server.database.]type_name lib_name entry_point [c++if_major.c++if_minor]
```

Each line in the map file consists of:

1. The server type, optionally prefixed with the server and database name.
2. The name of the shared library. It can be qualified with a specific path. Otherwise the library is located from the environment variable LD_LIBRARY_PATH on Solaris or PATH on Windows™.
3. The entry point in that library for the factory routine for the object.
4. Optionally, the version of the C++ library for which an object was built, given in the format *major.minor*.

Within the lines, entries must be separated by tabs or spaces. For example:

```
myserver1.mydatabase.Polygon3D /home/myhome/lib3d.so _makePoly3D
Polygon3D lib3d.so _makePoly3D
```

The library does not attempt to instantiate an object if the major version of the library is different or if the minor version of the C++ library that an object was created for is higher than the minor version of the installed C++ library. Value object authors can use the IT_VERSION macro (defined in `itcppop.h`) to determine the version of the library an object is being built for. The server and database name can be used to specify the type name.

In the preceding example, the Object Interface for C++ library instantiates an object for Polygon3D by using the library from `/home/myhome` if the connection is made to **myserver.mydatabase**; otherwise it uses the second library.

The map file can have any valid file name. On UNIX™, the default map file is `$INFORMIXDIR/etc/c++map`. On Windows™, the default map file is `%INFORMIXDIR%\etc\c++map`. In addition, you can manually set the **INFORMIXCPPMAP** environment variable to the fully qualified path of the map file, including the name of the map file itself.

Type names that contain white space characters (or multibyte character strings) must appear in double quotation marks in the type map file. Double quotation marks inside the type names in the type map file must be duplicated.

The entry point is the C function that is called to create a type. Enter qualified type names before unqualified type names. The **INFORMIXCPPMAP** environment variable can have several map files separated by colons (`:`) on UNIX™ or semicolons (`;`) on Windows™. The `.so` extension on Solaris and `.dll` on Windows™ are optional for the library name, and you can omit the file extension so that the same map file can be used in multiple environments.

Guidelines

When building Object Interface for C++ applications, observe the following guidelines:

- **Linkage**: the shared object library factory routine must have C linkage, not C++ linkage. For example:

```
extern "C" ITValue *makePoly3D(ITMVDesc *mv);
```

- **Mapping changes**: if the map file changes after a client application has loaded a shared object library, the application must flush its in-core map and reload (by calling the ReloadMapFiles method of the **ITFactory** class).

Operation class reference

This section is an alphabetized reference that lists and describes the operation classes. Each class has an assignment operator and a copy constructor, which are not listed in the tables of methods.

The ITConnection class

Base class: ITErrorManager

Manages a connection to a database and the errors that can occur. The **ITConnection** class is used to open a connection to the database server and to manage error objects and transaction states.

Only one result set can be outstanding on a DataBlade® API connection. The encapsulates DataBlade® API connection serialization through check-out (with ITConnection::CheckOutConn()) and check-in (with ITConnection::CheckInConn()). The **ITQuery**, **ITStatement**, and **ITCursor** methods that perform server access (for example, ITQuery::ExecToSet() and ITCursor::Prepare()) check out the connection, perform the operation, and then check the connection in. ITQuery::ExecForIteration() checks out the connection for the duration of the results retrieval. Some operations (for example, large object operations and server routine execution) might require server access but do not affect the results set. These operations use ITConnection::GetConn() to get the DataBlade® API connection without checking it out.

Applications generally do not need to use the DataBlade® API connection directly. Value objects do not attempt to perform the operations that would require checking the connection out (it is likely to be checked out by the query object). Value objects can use the DataBlade® API connection obtained by calling ITConnection::GetConn() to perform the operations that do not require connection checkout. Value objects gracefully handle the possibility of ITConnection::GetConn() returning `NULL` (when the connection is not open).

The following table lists the methods provided by this class.

Method	**Description**
ITConnection()	Creates an unconnected connection object with the default DBInfo.
ITConnection(MI_CONNECTION *, enum ITTransactionState tstate)	Constructs a connection object for an existing connection and sets the transaction state with the provided argument.
ITBool SetDBInfo(const ITDBInfo &)	Sets the DBInfo of the connection without opening the connection. Returns `TRUE` if successful; `FALSE` if the connection is currently open.

Method	Description
ITBool Open(const ITDBinfo &db)	Opens the connection with the specified DBInfo.
ITBOOL Open()	Opens the connection with the default DBInfo.
ITBool Close()	Closes the database connection.
ITBool IsOpen() const	Returns `TRUE` if the connection is open, `FALSE` if it is not.
MI_CONNECTION *GetConn()	Returns DataBlade® API connection encapsulated by ITConnection object, `NULL` if ITConnection is not open.
ITBool SetTransaction(enum ITTransactionState, ITCallBackFuncPtr func=NULL, void *userdata=NULL)	Sets the transaction state. The transaction state can be set to Begin to begin the transaction, or Commit or Abort to finish it. See Connection transaction states on page 13 for more information. The CallBackFuncPtr and user data arguments are reserved for future use. The transaction states are: ITTransactionState::NONE ITTransactionState::AUTO ITTransactionState::BEGIN ITTransactionState::COMMIT ITTransactionState::ABORT
enum ITTransactionState GetTransactionState()	Returns the transaction state.
MI_CONNECTION *CheckOutConn()	Checks out the DataBlade® API connection handle in order to bypass this C++ interface. Returns `NULL` if the connection is already checked out or the ITConnection is not connected to a database. **Restriction:** This interface is for compatibility with DataBlade® API. Direct use of the DataBlade® API is discouraged.
ITBool CheckInConn()	Returns a checked-out DataBlade® API connection to the ITConnection. Returns `TRUE` if the connection was previously checked out, `FALSE` otherwise.
const ITConnectionStamp &GetStamp()	Gets the current connection stamp object. (For details, see The ITConnectionStamp class on page 50.)
const ITDBInfo &GetDBInfo()	Retrieves the DBInfo object information with which the connection was initialized.

The ITConnectionStamp class

Base class: ITObject

Connection events can invalidate value objects to which the application maintains references. A connection stamp can be extracted from a connection and compared to a previously extracted connection stamp to determine whether the connection object calls the same server connection and transaction. This object is intended primarily for the development of DataBlade® value objects. For more details, see Value objects and connection events on page 41.

Typically, a user object gets a connection stamp when it establishes a connection. Whenever the value object must verify that this transaction or connection is current, it gets another connection stamp and compares them using one of the comparison methods listed in the following table.

This class provides the following methods.

Method	Description
ITBool Equal(const ITConnectionStamp &) const	Indicates whether these stamps refer to the same connection and transaction.
ITBool EqualConnInstance(const ITConnectionStamp &) const	Indicates whether these stamps refer to the same connection instance.
ITBool EqualTransactionInstance(const ITConnectionStamp &) const	Indicates whether these stamps refer to the same transaction.

The ITContainerIter class

Base class: ITObject

Provides a simple, syntactically compact interface for extracting C++ base-type values (such as **int**, **long**, or **double**) from an object. Value objects passed to an **ITContainerIter** object must expose either an **ITContainer** or **ITContainCvt** interface.

This class provides the following methods.

Method	Description
ITContainerIter(ITContainer *), ITContainerIter(ITEssential *), ITContainerIter(ITContainCvt *)	Binds an ITContainer or ITContainCvt interface into the newly constructed iterator. The values in the object can later be extracted by using the >> operator.
ITContainerIter &operator >> (ITValue *&)	Extracts a pointer to the value interface of the next column. If there are no more values left, sets the ITValue pointer to `NULL`. This method can be used to extract the individual columns into interface pointer variables. The ITValue interface must be released by the application.

Method	Description
ITContainerIter &operator >> (*modifiable_ lvalue* &)	Copies the value into the specified variable. This operation raises an exception if the column and variable type are not compatible or convertible. Valid types for the *modifiable_ lvalue* parameter are as follows: short int double long float long double const char * ITString ITInt8 bool (if the C++ compiler supports bool)
ITContainerIter &ITContainerIter::operator<< (<type>)	Sets the value of a contained item from the value of the C++ type given as <type>, where <type> can be any of the following type specifiers: short int double long float long double const char * ITString & const ITString & ITInt8 bool (if the C++ compiler supports bool) ITContainerIter has a state that can be either StateOK, StateOutOfBounds, StateUninitialized, or StateConversionFailed. If ITContainerIter state is not StateOK, the use any of the operators does not perform any conversions and does not change the state or position in the container.
void Reset()	Resets the state to StateUninitialized or StateOK, depending on whether the container iterator was initialized.

Method	Description
StateCode State()	Retrieves the state of the container iterator. State might be one of the following: StateUninitialized, StateOK, StateOutOfBounds, or StateConversionFailed. The initial state of the ITContainerIter is StateUninitialized if the value object that ITContainerIter was created on does not expose ITContainCvt or ITContainer; otherwise the initial state is StateOK. Calling ITContainerIter::Reset() resets a state to this initial state. StateOutOfBounds is set by the shift operators (<< >>) when the item position exceeds the number of items in the container. StateConversionFailed is set by the operator if the container does not expose ITContainCvt and the item does not expose ITConversions, or if the conversion function fails.
int Index()	Retrieves the current container iterator index.

The ITCursor class

Base class: ITErrorManager

Manages database cursors.

This class provides the following methods.

Method	Description
ITCursor(const ITConnection &)	Creates a cursor object for a connection.
ITBool Prepare(const ITString &, int nargs = 0, const ITString *typeNames = 0, ITEssential **outerunkns = 0);	Prepare() prepares the SQL statement and creates a list of null-valued parameters. Prepare() takes as an argument an ITString object, which must be a single valid SQL statement. See The ITStatement class on page 71.
ITBool Drop()	Drops the prepared statement and removes the parameter list.
int NumParams() const	Returns the number of parameters in a prepared statement. It returns `-1` if the statement has not been successfully prepared.
ITValue *Param(int)	Allows the application to return the ITValue of a parameter . The argument is a zero-based parameter

Method	Description
	number. Param() returns NULL if there are no parameters or if the parameter number is out of bounds.
ITBool SetParam(int parmno, ITDatum *)	Sets the cursor parameter with the number equal to parmno to be the value object passed as the second argument. Returns TRUE if successful, FALSE if it fails. Supports binding parameters in both binary and text mode. For more information, see the example in Usage on page 73.
const ITString &QueryText() const	Returns the query text. Returns ITString::Null if the statement has not been successfully prepared.
ITBool IsReadOnly() const	Returns TRUE if the cursor is read only, otherwise returns FALSE.
ITBool Open(int flags = 0, const ITString &tableName = ITString::Null)	Opens a cursor with the flags taken from the sum of the Open() flag values. Flag values can be the sum of: ITCursor::Sensitive ITCursor::ReadOnly ITCursor::Scrollable ITCursor::Reopt ITCursor::Hold Calling Open() without arguments opens a nonscrollable cursor. Open() returns TRUE on success, FALSE otherwise. It is an error for a cursor to be both scrollable and can be updated. If updates are performed by using the cursor, tableName must be passed as the second argument.
ITBool Close()	Closes the cursor. After calling Close(), the application can modify parameters and open a new cursor.
const ITString &Command() const	Returns the command verb.
const ITString &Name() const	Returns the name of the cursor. Returns ITString::Null if the cursor has not been opened successfully.
ITRow *NextRow(ITEssential **outerunkn = NULL, enum ITPosition pos = ITPositionNext, long jump = 0)	Fetches the next row and returns the pointer to the ITRow interface of the row object. Returns NULL if the row cannot be fetched. Until the cursor row is modified or deleted, a new instance of that row can be fetched again by specifying fetch position ITPositionCurrent even if the cursor is not scrollable.

Method	Description
ITBool UpdateCurrent()	Executes the SQL statement UPDATE WHERE CURRENT OF using the values of the updated columns in the current row. Returns `TRUE` if the update was successful and `FALSE` if it was not. It is an error for the application to call UpdateCurrent() if NextRow() or a fetch function fails.
ITBool DeleteCurrent()	Executes the SQL statement DELETE WHERE CURRENT OF. Returns `TRUE` if the deletion was successful and `FALSE` if it was not. It is an error for the application to call DeleteCurrent() if NextRow() or a fetch function fails.
ITValue *Fetch(ITEssential **outerunkn = NULL, enum ITPosition pos = ITPositionNext, long jump = 0)	Fetches a row from the cursor and returns the pointer to its ITValue interface.
const ITTypeInfo *RowType() const	Returns server type information about the row to be fetched. Can be called after Prepare() to get row type information before opening the cursor.
ITBool IsScrollable() const	Returns `TRUE` if the cursor is opened as scrollable, otherwise returns `FALSE`.

Usage

ITCursor can pass binary data as parameters in prepared SQL SELECT statements. In addition, ITStatement can pass binary data as parameters in prepared SQL DML statements DELETE, INSERT, UPDATE, and SELECT. For an example showing how can be used to set a parameter to binary data in a prepared INSERT statement, see Usage on page 73

The ITDBInfo class

Base class: ITErrorManager

Sets or returns information about connections to HCL Informix® databases (such as the user, database, system, and password). When an **ITDBInfo** is used to open a connection, the **ITDBInfo** becomes frozen and cannot be modified by using the **Set** calls.

This class provides the following methods.

Method	Description
ITDBInfo()	Constructs an ITDBInfo object for the system environment of the user.
ITDBInfo(const ITDBInfo &)	Copy constructor. The ITDBInfo copy constructor makes a deep copy rather than a shallow copy. The new ITDBInfo

Method	Description
	object is thawed and can be modified by using the Set calls.
ITDBInfo(const ITString &db, const ITString &user = ITString(), const ITString &system = ITString(), const ITString &passwd = ITString());	Constructs ITDBInfo and sets system database and user information. This method has these parameters: *db* is the database name. *user* is the user name. *system* is the system name. *passwd* is the password.
ITBool operator==(const ITDBInfo &) const;	Compares the instances of the **ITDBInfo** objects.
ITBool Frozen() const	Returns `TRUE` if the information of this database object is frozen, or `FALSE` if the information is not frozen.
ITBool Freeze()	Freezes the database information of the object.
ITBool CreateDatabase (int flags = ITDBInfo::Default, const ITString &dbspace = ITString::Null)	Creates the database; returns `TRUE` if the database was successfully created, `FALSE` if it was not. The database name and server name are taken from ITDBInfo. The following values are valid for type: ITDBInfo::Default ITDBInfo::Log ITDBInfo::BufferedLog ITDBInfo::ANSIModeLog *dbspace* is the name of dbspace; default dbspace if omitted.
ITBool DropDatabase()	Drops the database; returns `TRUE` if the database was successfully dropped, `FALSE` if it was not.
ITBool SetUser(const ITString &)	Sets the user name.
ITBool SetDatabase(const ITString &)	Sets the database name.
ITBool SetSystem(const ITString &)	Sets the system name.
ITBool SetPassword(const ITString &)	Sets the password.
const ITString &GetUser() const	Returns the user name.
const ITString &GetDBLocaleName() const	Returns the database locale name.
const ITString &GetSystem() const	Returns the system name.
const ITString &GetDatabase() const	Returns the database name.

The ITDBNameList class

Base class: ITErrorManager

Encapsulates the list of database names. Obtain the list by calling the Create() function. After the list is created, applications can use NextDBName() and PreviousDBName() to traverse it.

This class provides the following methods.

Method	Description
ITDBNameList()	Creates an instance of **ITDBNameList**.
ITBool Create()	Creates a list of all databases for all systems in **DBPATH** and **INFORMIXSERVER**.
ITBool Create(const ITString &)	Creates a list of all databases for a system with the specified name.
ITBool Create (ITConnection &)	Creates a list of all databases corresponding to the connection.
ITBool IsDBName(const ITString &)	Returns `TRUE` if the name supplied as an argument appears in the database name list; `FALSE` if it does not.
const ITString &NextDBName()	Returns the reference to the next database name; returns ITString::Null if there is no next database name.
const ITString &PreviousDBName()	Returns the reference to the previous database name; returns ITString::Null if there is no previous database name.
void Reset()	Resets the database list name to the state it was in immediately after the list was created.

The ITErrorManager class

Base class: ITObject

Manages error callbacks from the server or from the client library. Multiple callbacks can be set on an **ITErrorManager** instance. **ITErrorManager** defines functionality used by a number of subclasses for managing and dispatching errors for different operations, such as issuing queries and retrieving results. Using the **ITErrorManager** class, applications can set callback functions to be triggered by exceptional conditions generated during database access.

Events that might trigger the call to callback functions are:

- Server exceptions-SQL errors, transaction state changes, warnings, and other exceptions.
- DataBlade® API library exceptions.
- C++ library events.

Callback functions must have the following signature:

```
typedef void      (*ITCallBackFuncPtr)
                  (const ITErrorManager &errorobject,
                   void *userdata,
                   long errorlevel);
```

The *userdata* parameter is for data passed to the callback function. The error-level parameter corresponds to the DataBlade® API error level, and indicates whether the error is a message, an exception, or an unrecoverable error.

This class provides the following methods.

Method	**Description**
ITBool Error() const	Returns `TRUE` if either a server or client error occurs.
const ITString &SqlState() const	Returns the SQLSTATE code of an error. For details about SQLSTATE, see the *Informix® Guide to SQL: Syntax*.
const ITString &ErrorText() const	Returns error message text.
ITBool AddCallback(ITCallbackFuncPtr userfunc, void *userdata)	Adds a callback. For details, see Implementation notes on page 7.
ITBool DelCallback(ITCallbackFuncPtr userfunc, void *userdata)	Deletes a user-defined callback registered through AddCallback().
ITBool DispatchErrorText(const ITString &message)	Dispatches an error message with the specified message text.
ITBool Warn() const	Returns `TRUE` if a warning occurred.
const ITString & WarningText() const	Returns warning message text.

The ITFactoryList class

Base class: none

This functionality provided by this class is only supported with Informix® databases.

Adds mappings from Informix® data types to functions that build value objects to represent instances of these data types. For more details, see Build simple value objects on page 33.

Developers of value objects can either use this class and compile the value object code into applications or, for greater reusability, use dynamic loading as described in Dynamic loading on page 47.

This class provides the following methods.

Method	Description
ITFactoryList(const char *name, ITFactoryFuncPtr func, ITBool flushable = false);	Declares a mapping from the specified server type (the *name* parameter) to the specific factory function pointer (*func*).
static void ReloadMapFiles(ITErrorManager *errobj);	Forces a reload of the factory object map files. The map files map server types to dynamically loadable libraries that contain functions for building value objects. If the map changes, an application can call this procedure to reload the maps.
static ITBool FlushDynamicFactories(ITErrorManager *errobj);	Unloads all the dynamically loaded libraries and clears dynamic entries from the list of factories To retain the ability to scan the map files after dumping, applications call ReloadMapFiles() instead of FlushDynamicFactories.
static void Init()	Initializes the built-in factory list in case the compiler does not perform this initialization automatically.
static ITValue *DatumToValue (ITMVDesc &)	Creates the instance of the value object by using the provided ITMVDesc. Returns the pointer to the ITValue interface of the created object. Returns `NULL` if it fails. In the absence of the factory for the constructed type, DatumToValue() uses the factory of the constructor. For example, it would use the built-in set factory for the type set (integer not null).
GetInitState()	Verifies that the library loaded into memory is properly initialized. For more information and an example, see Successful initialization verification on page 59.

Successful initialization verification

Under some circumstances, the library might be loaded into memory but not properly initialized. For example, if the environment variable **CLIENT_LOCALE** is set to an invalid locale, the GLS library does not properly initialize, and thus the Object Interface for C++ library also does not properly initialize.

To allow Object Interface for C++ application programs to verify that initialization succeeded, several new members have been added to the ITFactoryList class (defined in the public header file `$INFORMIXDIR/incl/c++/itcppop.h`):

```
class IT_EXPORTCLASS ITFactoryList
    {
        ...

    public:

     // These are the built-in factory list initialization state values
      enum InitState { NOTINIT, INITING, INITED };
    // This function can be used to determine if the built-in factory
    // list initialized properly. It returns
    ITFactoryList::NOTINIT
    // if initialization failed.
    static InitState GetInitState();

     ...

    };
```

The user application calls GetInitState() before using any Object Interface for C++ classes or interfaces to determine if Object Interface for C++ initialized properly, as follows:

```
main(int, char *)
      {
         // check that Object Interface for C++ DLL initialized ok
         if (ITFactoryList::GetInitState() == ITFactoryList::NOTINIT)
         {
            cout << "Error: Object Interface for C++ DLL not
            initialized" <<
                           endl;
            cout << "Error: exiting program" << endl;
            return -1;
         }
       ...
```

The ITInt8 class

Base class: none

Encapsulates 8-byte integer value. This class can be used in any client application, although only HCL Informix® supports the **int8** data type.

This class provides the following methods.

Method	**Description**
ITInt8()	Creates uninitialized instance of ITInt8.
ITInt8 &operator=(<<type>)	Sets ITInt8 to the value of *<type>*, where *<type>* is one of the following:

Method	Description
	int long float double mi_int8 IT_LONG_LONG where IT_LONG_LONG is a compiler-provided 8-byte integer (if any). The result of the conversion might not fit into the type specified by *<type>*.
IsNull()	Returns `TRUE` if an object does not represent a valid 8-byte integer.
Conversion operators	ITInt8 provides conversions to the value of one of the following types: int long float double mi_int8 ITString IT_LONG_LONG
Other operators	ITInt8 provides assignment comparison, and arithmetic operators. The results of arithmetic operations on ITInt8 objects might not fit into 8 bytes, in which case, the result would not be a valid ITInt8 object.

In Version 2.70, you can use new constructors to create objects by using each of the built-in numeric types as initialization arguments. This eliminates the need to explicitly assign a numeric type that is not an int8 (for example, int) to an ITInt8 object before comparing it with an ITInt8 object.

The new constructors are:

```
ITInt8( const int );
    ITInt8( const long );
    ITInt8( const float );
    ITInt8( const double );
    ITInt8( const mi_int8 );
    ITInt8( const ITString & );
    #ifdef IT_COMPILER_HAS_LONG_LONG
    ITInt8( const IT_LONG_LONG );
    #endif
```

Before version 2.70, to initialize an ITInt8 object, the application must assign a value to an ITInt8 object by using the assignment operator (=), as follows:

```
int i = 100;
     ITInt8 i8;
     i8 = i;
     if ( i8 == (ITInt8)i )
```

With Version 2.70 and later, the assignment can be replaced by an ITInt8 constructor call:

```
int i = 100;
     ITInt8 i8(i); // or ITInt8 i8(100);
     if ( i8 == (ITInt8)i )
```

The ITLargeObjectManager class

Base class: ITErrorManager

This functionality provided by this class is only supported with HCL Informix® databases.

Manipulates large objects. Large object operations are similar to normal file management operations (read, write, seek, and other operations). Client value objects based on large objects in the server typically expose an **ITLargeObject** interface. For details, see Object Containment and Delegation on page 44. See the *Informix® Guide to SQL: Reference* for details about large objects.

This class provides the following methods.

Method	**Description**
ITLargeObjectManager(const ITConnection &)	Creates a large object manager for the specified connection.
ITBool SetHandleText(const ITString &handleText, int flags = MI_LO_RDWR)	Sets a manager to handle a large object, where *const* ITString is the large object handle in text format.
const ITString &HandleText()	Returns the handle of the currently managed large object in a text format.
ITBool SetHandle(const MI_LO_HANDLE *handle, int flags = MI_LO_RDWR)	Sets a manager to handle a large object, where *const MI_LO_HANDLE* is a pointer to the large object handle.
const MI_LO_HANDLE *Handle()	Returns the handle of the currently managed large object in the binary format through the constant *MI_LO_HANDLE*.
int Read(char *buf, int cnt)	Reads bytes from the large object at the current position.
int Write(const char *buf, int cnt)	Writes bytes to the large object at the current position.
ITInt8 Seek(ITInt8 off, int cntl = 0)	Sets the current position of the large object; cntl is a position like UNIX™ lseek (0 is absolute position, 1 is

Method	Description
	relative to current position, and 2 is relative to end of the large object).
ITInt8 Size()	Returns the total size of the large object.
ITBool SetSize(ITInt8)	Sets the total size of the large object.
ITBool CreateLO(int flags = IT_LO_WRONLY | IT_LO_APPEND)	Creates a large object. Sets the handle of the manager to the new large object. The handle is then inserted into a table column (for example, by using a prepared SQL **insert** statement).
ITBool CreateLO(MI_LO_SPEC*, int flags = IT_LO_WRONLY | IT_LO_APPEND)	Creates a large object with the specifications provided.
ITBool Close()	Closes the smart large object managed by this ITLargeObjectManager instance. Returns `TRUE` if the smart large object was not open or was closed successfully. Returns `FALSE` on failure.

Accessing smart large objects in nondefault sbspaces

One way to access smart large objects in nondefault spaces is to call the client-side DataBlade® API functions that create, initialize, and set the column-level characteristics of a large object specification structure and then pass a pointer to this structure (**MI_LO_SPEC *LO_spec**) to the overloaded function.

```
ITBool CreateLO( MI_LO_SPEC *LO_spec,
                     int flags=IT_LO_WRONLY | IT_LO_APPEND) ;
```

A better way is to introduce a new C++ class to encapsulate a large object specification structure and possibly modify the existing **ITLargeObjectManager** class to support passing the column-level storage characteristics of smart large objects as encapsulated C++ objects for use by ITLargeObjectManager::CreateLO.

Here is a description of the short-term solution. Before calling **CreateLO**, the following DataBlade® API call sets the fields of **LO_spec** to the column-level storage characteristics of column **dolmdolm1.testdata**, which is the CLOB column:

```
res = mi_lo_colinfo_by_name(miconn,
                                  (const char *)"dolmdolm1.testdata",
                                  LO_spec);
```

Among the attributes for column testdata is the sbspace location specified by the PUT clause when table **dolmdolm1** is created. The smart large object location attribute is used by **CreateLO** (which calls DataBlade® API function mi_lo_create()) when it creates the smart large object.

Here is the complete, modified test case with the new solution:

```
>>>>>>>>>>>> Begin modified test case with new solution >>>>>>>>>>>>
#include <stdlib.h>
#include <iostream.h>
```

```
#include <it.h>
int
main(int argc, const char *argv[])
{
    ITDBInfo dbinfo;
    ITConnection conn;
    char buf[1024];
    int i;
    ITString types[2];
    ITString sqlcmd;
    types[0] = "VARCHAR";
    types[1] = "VARCHAR";
    cout << " INFORMIXSERVER   : ";
    cin.getline(buf, sizeof(buf));
    if (!dbinfo.SetSystem(buf)){
            cout << "Could not set system " << endl;
        return (1);
    }
    cout << " DATABASE : ";
    cin.getline(buf, sizeof(buf));
    if (!dbinfo.SetDatabase(buf)){
            cout << "Could not set database " << endl;
            return (1);
    }
    cout << " USER     : ";
    cin.getline(buf, sizeof(buf));
    if (!dbinfo.SetUser(buf)){
            cout << "Could not set user " << endl;
            return (1);
    }
    cout << " PASSWORD : ";
    cin.getline(buf, sizeof(buf));
    if (!dbinfo.SetPassword(buf)){
            cout << "Could not set password " << endl;
            return (1);
    }
    if (!conn.Open(dbinfo) || conn.Error()) {
    if (!conn.Open() || conn.Error()) {
            cout << "Could not open database " << endl;
            return (1);
    }
    cout << "Start Transaction ..." << endl;
    if (!conn.SetTransaction(ITConnection::Begin)) {
            cout << "Could not start transaction " << endl;
            return (1);
    }

   ITStatement stmt(conn);
    cout << " SBLOBSPACE : ";
    cin.getline(buf, sizeof(buf));
    sqlcmd = "create table dolmdolm1 (";
    sqlcmd.Append("uid integer primary key,");
    sqlcmd.Append("testdata CLOB)");
    sqlcmd.Append(" PUT testdata in (");
    sqlcmd.Append(buf);
    sqlcmd.Append(") lock mode row;");
    cout << sqlcmd << endl;
    if (stmt.Error()) {
```

```
        cout << "Could not create statement " << endl;
        return (1);
}
if (!stmt.Prepare(sqlcmd)) {
        cout << "Could not prepare create statement " << endl;
        return (1);
}
if (!stmt.Exec())  {
        cout << "Could not execute create statement " << endl;
        return (1);
}
if (!stmt.Drop()) {
        cout << "Could not drop create statement " << endl;
        return (1);
}
cout << "Please monitor your sblobspaces, [return to continue]";
cin.getline(buf, sizeof(buf));
/************* begin new solution code *************************/
MI_LO_SPEC *LO_spec = NULL;
MI_CONNECTION *miconn = NULL;
mi_integer res;
ITLargeObjectManager lo(conn);

miconn = conn.GetConn();
if (miconn != NULL)
{
   res = mi_lo_spec_init(miconn, &LO_spec);
   if (res == MI_ERROR)
   {
      cout << "stmt_test: mi_lo_spec_init failed!" << endl;
      return (1);
   }
   res = mi_lo_colinfo_by_name(miconn,
                               (const char *)"dolmdolm1.testdata",
                               LO_spec);
   if (res != MI_ERROR)
   {
       cout << endl << "Create a large object. Please wait ..." <<
       endl;

       ITBool status = false;
       status = lo.CreateLO(LO_spec, IT_LO_WRONLY | IT_LO_APPEND);
       if (status = true)
       {
          for (i = 0; i < 1000; i++)
              lo.Write("12345678901234567890123456789012345678901234567890
              01234567890123456789012345678901234567890123456789",100);
       }
       else
   {
          cout << "stmt_test: CreateLO w/non-default sbspace
          failed!" <<
          endl;
          return (1);
       }
   }
   else
   {
```

```
          cout << "stmt_test: mi_lo_colinfo_by_name failed!" << endl;
          return (1);
       }
    }
    else
    {
       cout << "stmt_test: conn.GetConn returned NULL!" << endl;
       return (1);
    }
    /************* end new solution code *************************/
    cout << "The default sblobspace has changed" << endl;
    cout << "Please monitor your sblobspaces, [return to continue]";
    cin.getline(buf, sizeof(buf));

    cout << endl << "inserting row into dolmdolm1" << endl;
    if (!stmt.Prepare("insert into dolmdolm1 values (?,?);",2,types))
    {
            cout << "Could not prepare insert cursor " << endl;
            return (1);
    }
    ITValue *param;
    param = stmt.Param(0);
    param->FromPrintable("0");
    param->Release();
    param = stmt.Param(1);
    param->FromPrintable(lo.HandleText());
    param->Release();
    if (!stmt.Exec())  {
            cout << "Could not execute insert statement " << endl;
            return (1);
    }
    if (!stmt.Drop()) {
            cout << "Could not drop insert statement " << endl;
            return (1);
    }
    cout << endl;
    cout << "Please monitor your sblobspaces." << endl;
    cout << "The large object is still stored within the default
    sblobspace." << endl;
    cout << "[return to continue]";
    cin.getline(buf, sizeof(buf));
   /*
    cout << "Rollback Transaction ..." << endl;
    if (!conn.SetTransaction(ITConnection::Abort)) {
            cout << "Could not rollback transaction " << endl;
            return (1);
    }
   */
    cout << "Commit Transaction ..." << endl;
    if (!conn.SetTransaction(ITConnection::Commit)) {
            cout << "Could not commit transaction " << endl;
            return (1);
    }
    conn.Close();
    cout << endl;
    return 0;
}
>>>>>>>>>>>> End modified test case with new solution >>>>>>>>>>>>>>>
```

The ITMVDesc class

The **ITMVDesc** structure is not an operation class, but a descriptor that holds the instance information necessary to create a value object. The **ITMVDesc** structure is passed to the factory constructor function when an object of a given server type is retrieved from the server and loaded into the application.

This structure contains the following individual members.

Member	Description
long vf_datalength	Data length in bytes, pointed to by the member data pointer vf_data.
ITConnection *vf_connection	Pointer to the connection object.
int vf_libmivaluetype	Return value of the call to a DataBlade® API function call [mi_value(...)]; see the *Informix® DataBlade® API Programmer's Guide* for complete documentation of DataBlade® API function calls.
char *vf_data	Points to the datum underlying the value object. For example, for the server type lvarchar, vf_data points to the MI_LVARCHAR structure.
ITypeInfo *vf_origtypeinfo	Points to the ITTypeInfo object for the value object.
ITEssential *vf_outerunknown	Points to the IUnknown interface of the object that is the controlling unknown for the object delegation/aggregation process. Value is `NULL` if there is no controlling unknown. The vf_outerunknown member is assigned the value of the inner unknown of the object when ITMVDesc * is passed to the entry point function MakeValue(ITMVDesc *), which is implemented by the value object developer.
ITPreserveData *vf_preservedata	Can point to the ITPreserveData interface of an object that manages the datum memory. For a detailed description of the vf_preservedata member and its use, see Value objects and connection events on page 41.

The ITObject class

Base class: none

A common base class that serves solely as an abstraction of an object. Instances of operation interface classes (except for the **ITString** class) are all derived from the **ITObject** class.

This class provides the following methods; all operation classes override these methods to perform reference counting for copy operations and assignment.

Method	Description
virtual ~ITObject()	Virtual destructor.
ITObject &operator=(const ITObject &)	Assignment operator.

The ITPosition class

ITPostion is an enumerated type.

Functions that might perform positioning (for example, ITCursor::NextRow() and ITSet::Fetch()) accept an instance of **ITPosition** as one of their arguments.

Field	Description
ITPositionCurrent	Specifies the current position in the sequence.
ITPositionNext	Specifies the next position in the sequence.
ITPositionPrior	Specifies the previous position in the sequence.
ITPositionFirst	Specifies the first position in the sequence.
ITPositionLast	Specifies the last position in the sequence.
ITPositionAbsolute	Specifies that the corresponding (always positive) offset is from the beginning of the sequence; for example: `value = set.Fetch(0, ITPositionAbsolute, 10)`
ITPositionRelative	Specifies that corresponding offset is from the current position; for example: `value = list.Fetch(0, ITPositionRelative, -1)`

The ITPreserveData class

Base class: none

Provides an interface for maintaining a reference to database data received from the server, for use by the implementor of a value object. For details, see Value objects and connection events on page 41.

This class provides the following methods.

Method	Description
virtual unsigned long AddRef()	Increment reference count.
virtual unsigned long Release()	Decrement reference count.

The ITQuery class

Base class: ITErrorManager

Manages query processing, including errors that occur as a result of queries. **ITQuery** is derived from **ITErrorManager**. Results are returned as binary data encapsulated by value objects. To obtain a text version of the results, you must use conversion methods such as ITValue::Printable.

For details about using the different query methods, see When to use the different ITQuery methods on page 15.

For the ExecOneRow, ExecToSet, and NextRow methods, the *unknwn* argument is the address of a pointer to an **ITEssential** interface of an object that will be the parent of any subobjects that might be created by the method. The newly created subobject returns its own **ITEssential** interface pointer in the same argument (which is an argument of type in/out) if the object delegation was successful. The subobject reference count is 1 after the call. The default argument is `NULL` to indicate that no object delegation is to be performed.

An **ITQuery** is always created in the context of the server connection.

The ITQuery::ExecOneRow method returns `NULL` if an error occurred, but also returns `NULL` if the query returns no rows but is not in error. To check if there was a DBMS error, use the Error method.

This class provides the following methods.

Method	Description
ITQuery(const ITConnection &)	Constructor.
ITBool ExecForStatus(const ITString &)	Issues a query for which the caller is only interested in result status such as whether the query succeeded, the number of rows affected, and results. No result rows are returned. Specify the query in the ITString parameter. ExecForStatus() takes as an argument an ITString object, which must be a single valid SQL statement.
ITRow *ExecOneRow(const ITString &, ITEssential **unknwn = NULL)	Issues a query for which a single result row is expected and returned. Returns a null pointer if an error occurs. If the query returns more than one row, the additional rows are discarded and no error is returned. Specify the query in the ITString parameter. ExecForStatus() takes as an argument an ITString object, which must be a single valid SQL statement.

Method	Description
ITSet *ExecToSet(const ITString &, ITEssential **unknwn = NULL)	Issues a query and returns results by using a rowset object that has an ITSet interface. Returns a null pointer if an error occurs. Specify the query in the ITString parameter. ExecForStatus() takes as an argument an ITString object, which must be a single valid SQL statement.
ITBool ExecForIteration(const ITString &)	Issues a query. Returns `TRUE` if the query was accepted, `FALSE` if an immediate problem (such as a syntax error) was found. If the query is accepted, this call returns `TRUE`, and the user must call NextRow to fetch the results in order. NextRow must be called repeatedly until it returns `NULL` (meaning all rows are read) before your application can issue another query or perform other database operations on the connection. Specify the query in the ITString parameter. ExecForStatus() takes as an argument an ITString object, which must be a single valid SQL statement.
long RowCount()	Returns the number of rows affected by the last query issued on the ITQuery.
const ITString &Command()	Returns the type of SQL statement (select, create, update, and other statements).
ITRow *NextRow(ITEssential **unknwn = NULL)	Returns the next result row, if any. Used with ExecForIteration to process results. Returns `NULL` when the last result row has been returned. If a query was issued with ExecForIteration, the RowCount and Command methods are not valid until NextRow returns `NULL`. The result row value must be released when done. The underlying connection remains checked out until the last row is received.
const ITTypeInfo *RowType()	Returns server type information about the row that will be fetched. Used with ExecForIteration to get the type of results before actually getting the first row.
const ITString &QueryText()	Returns the text of an SQL query.
ITBool Finish()	Finishes processing the query results without retrieving all rows. Use this method with ExecForIteration to terminate a query without retrieving all the resulting rows.

The ITRoutineManager class

Base class: ITErrorManager

This functionality provided by this class is only supported with HCL Informix® databases.

The **ITRoutineManager** class provides an alternative way to execute server routines. When using **ITRoutineManager**, a connection does not have to be checked out to get or execute a routine (and a value object, therefore, can use it), and the execution of the routine commences faster (since there is no SQL to parse). See The ITConnection class on page 49 for information about connection checkout.

This class provides the following methods.

Method	Description
ITRoutineManager(ITConnection &)	Creates a Routine Manager for the specified connection.
ITBool GetRoutine(const ITString & signature)	Gets the descriptor for the registered routine from the server so the routine can be executed later by ExecForValue(). Returns `TRUE` if it gets the routine descriptor, `FALSE` otherwise.
const ITTypeInfo *ResultType() const	Returns a pointer to an ITTypeInfo instance that encapsulates the type of the return value of the routine. It returns `NULL` if did not get the routine.
int NumParams() const	Returns the number of parameters the routine accepts, `-1` if did not get the routine.
const ITTypeInfo *ParamType(int paramno) const	Returns a pointer to an ITTypeInfo instance that encapsulates the type of the specified parameter. It returns `NULL` if did not get the routine or if the argument is out of bounds.
ITValue *Param(int paramno) const	Returns a pointer to the parameter value object, `NULL` if did not get the routine or if the argument is out of bounds.
ITBool SetParam(int paramno, ITDatum *pdatum)	Sets the parameter value object for a specified parameter index to the pdatum. Returns `TRUE` on success, `FALSE` if did not get the routine or if the parameter is out of bounds, or the ITDatum is not of the same type as the corresponding routine parameter type.
ITValue *ExecForValue(ITEssential **outerunkn = NULL)	Executes the routine with the set parameters. Returns a pointer to the ITValue interface of the value object, instantiated for the return value. Returns `NULL` if did not get the routine or if execution failed.

The ITStatement class

Base class: ITErrorManager

The **ITStatement** class provides support for the execution of prepared queries that return no rows. For information about the use of prepared statements, see Using prepared statements on page 17.

This class provides the following methods.

Method	Description
ITStatement (const ITConnection &)	Creates an ITStatement object for the specified connection.
ITBool Prepare(const ITString &, int nargs = 0, const ITString *typeNames = NULL, ITEssential **outerunkns = 0)	Prepare() prepares the statement and creates a list of null-valued parameters. Prepare() takes as an argument an ITString object, which must be a single valid SQL statement. The names of the server types of the parameters that will be created can be supplied as an array of ITStrings. If an application does not provide parameter type names, this method uses parameter types communicated by the server. In the cases when the server does not communicate parameter types (as with UPDATE and DELETE queries) and they are not provided by the application, all parameters are created of the server type varchar(256). The application can provide an array of outer unknown pointers for delegation. After the call to Prepare(), elements of the outer unknowns array (if it was provided) are set to the inner unknowns. If the application provides either type names or outer unknowns, it must set the nargs parameter to their number.
ITBool SetParam(int parmno, ITValue *)	Sets the statement parameter with the number equal to parmno to be the value object passed as the second argument. Returns `TRUE` if successful, `FALSE` if it fails. The previous parameter object is released. Supports binding parameters in both binary and text mode. For more information, see the example in Usage on page 73.
int NumParams() const	Returns the number of parameters in a prepared statement. It returns `-1` if the statement has not been successfully prepared.
ITValue *Param(int)	Allows the application to return a ITValue of a parameter. The argument is a zero-based parameter number. Parm() returns `NULL` if there are no parameters or if the parameter number is out of bounds.
const ITString &Command() const	Returns an SQL command verb. Returns ITString::Null if the statement has not been successfully prepared.
const ITString &QueryText() const	Returns the query text. Returns ITString::Null if the statement has not been successfully prepared.

Method	Description
ITBool Exec()	Executes a prepared statement with the current parameter values. Returns `TRUE` if the execution was successful, `FALSE` if it was not. If the query returns rows, Exec() discards them.
long RowCount() const	Returns the number of rows affected by the last execution. Returns `-1` if the statement has not been executed.
ITBool Drop()	Drops the prepared statement and removes the parameter list.

Usage

ITStatement can pass binary data as parameters in prepared SQL DML statements DELETE, INSERT, UPDATE, and SELECT. In addition, SQL SELECT statements with parameters can be executed by using class ITCursor.

The following example shows how can be used to set a parameter to binary data in a prepared INSERT statement. The example uses the table CUSTOMER in the demonstration database STORES7:

```
#include <it.h>
#include <iostream.h>

int main()
{
    ITDBInfo db("stores7");
    ITConnection conn(db);

    conn.Open();

    if( conn.Error() )
    {
        cout << "Couldn't open connection" << endl;
        return -1;
    }
    ITQuery query( conn );
    ITRow *row;
    // Create the value object encapsulating the datum of SQL type CHAR(15)
    // by fetching a row from the database and calling ITRow::Column()
    if( !(row = query.ExecOneRow( "select lname from customer;" )) )
    {
        cout << "Couldn't select from table customer" << endl;
        return -1;
    }
    ITValue *col = row->Column( 0 );
    if( !col )
    {
        cout << "couldn't instantiate lname column value" << endl;
        return -1;
    }
    row->Release();
    ITDatum *datum;
    col->QueryInterface( ITDatumIID, (void **)&datum );
    if( !datum )
    {
        cout << "couldn't get lname column datum" << endl;
        return -1;
```

```
    }
    col->Release();
    // Prepare SQL INSERT statement, set the parameter to the value object that
    // encapsulates lname column value and execute INSERT
    ITStatement stmt( conn );
    if( !stmt.Prepare( "insert into customer (lname) values ( ? );" ) )
    {
        cout << "Could not prepare insert into table customer" << endl;
        return -1;
    }
    if( !stmt.SetParam( 0, datum ) )
    {
        cout << "Could not set statement parameter" << endl;
        return -1;
    }
    if( !stmt.Exec() )
    {
        cout << "Could not execute the statement" << endl;
        return -1;
    }
    return 0;
}
```

The ITString class

Base class: none

The **ITString** class is a minimal C++ string class that meets the needs of the . An **ITString** object created without any parameters is, by default, null-valued. All null-valued **ITString** objects are equal.

This class provides the following methods.

Method	**Description**
ITString()	Constructs a null string.
ITString(const char *str)	Constructs a string from null-terminated characters. This method assumes that the contents of the *str* buffer are in the client code set.
ITString(const char *str, ITBool in_server_codeset)	Constructs a string from null-terminated characters. The *in_server_codeset* parameter specifies whether the buffer is in the server code set.
operator const char *() const const char *Data() const	Returns a pointer to null-terminated characters of the value of the string or `NULL`. Do not delete this returned value.
int Length() const	Returns the number of multibyte characters in a string, excluding termination characters.

Method	Description
int Size() const	Returns the number of bytes in a string.
ITString &Trim(const char *pmc)	Trims the single, rightmost occurrence of the character (not string) starting at `c` within the string encapsulated by an ITString object. Trim() encapsulates searching for the rightmost character (which can be a multibyte byte character in a given locale) of the encapsulated string and truncation of that character. The search is performed by calling ITLocale::MRScan, which in turn encapsulates calling the GLS API function ifx_gl_mbsrchr().
ITString &TrimWhite()	Removes trailing white space.
ITBool Equal(const ITString &) const ITBool Equal(const char *) const	Compares this string with another. White space is significant.
ITBool EqualNoCase(const ITString &) const ITBool EqualNoCase(const char *) const	Compares one string with another. Case is not significant.
ITBool LessThan(const ITString &) const	Compares this string with another. White space is significant.
long Hash() const	Returns a long integer value suitable for determining a hash bucket by using modulo operations.
ITString &Append(const ITString &)	Appends a copy of another string to this string.
ITString &Append(const char *)	Appends a copy of the character string to this string.
ITString GetToken(int &) const	Gets the token from the string beginning with the position specified by the integer argument. *Token* is a quoted string, number, sequence of non-blank alphanumeric characters, or any other character. Argument is set to the position after the token.
ITBool IsQuoted() const	Returns `TRUE` if the string is in single or double quotation marks, `FALSE` otherwise.
ITBool Unquote()	If the string is quoted, removes the outer quotations and returns `TRUE`, otherwise returns `FALSE`.
const char *Scan(const char *) const	Returns a pointer to the first occurrence in the string buffer of the specified multibyte character.
static const ITString Null	Represents null string.

Method	Description
inline ITBool IsNull() const	Returns TRUE if string is null.
int operator<opname>(const ITString &, const ITString &)	Compares the two strings. The operators you can use for *opname* are: ==, !=, <, <=, >, >=.

The ITSystemNameList class

Base class: ITErrorManager

This class creates the system name list from the UNIX™ `sqlhosts` file or from the Windows™ registry entry under the `HKEY_LOCAL_MACHINE\Software \Informix\SqlHosts` key. After you create the system name list, you can traverse it with the NextSystemName() and PreviousSystemName() methods.

This class provides the following methods.

Method	Description
ITSystemNameList()	Constructs an ITSystemNameList object.
ITBool Create()	Creates the system name list from the sqlhosts file (on UNIX™) or from the registry entry under the `HKEY_LOCAL_MACHINE\Software\Informix\sqlhosts` key (on Windows™).
ITBool IsSystemName(const ITString &)	Returns TRUE if the name supplied as an argument appears in the system name list; FALSE if it does not.
const ITString &NextSystemName()	Returns the reference to the next system name; returns ITString::Null if there is no next system name.
const ITString &PreviousSystemName()	Returns the reference to the previous system name; returns ITString::Null if there is no previous system name.
void Reset()	Resets the system name list to the state analogous to the one it was in immediately after the list was created.

The ITTypeInfo class

Base class: ITObject

Contains information about the type of a value object as it exists in the database. **ITTypeInfo** identifies the types in the database that correspond to the C++ types that represent the values in the application. The **ITTypeInfo** class is also used to retrieve type information for values in a result set, and is essential for implementing user-defined value objects.

The **ITTypeInfo** class can be used to obtain a type name (unless the type is transient) and indicates whether the value is simple, row, or collection. A transient data type is a type that only lasts for the duration of an SQL statement. For example, in the following query:

```
create table foo (a int, b int, c int);
select * from (select a, b from foo);
```

The subquery (`select a, b from foo`) is a transient type that is a set of type row with two columns, a and b. This type is not persistent because it is devised by HCL Informix® to return the results of the SQL statement.

Simple types (types that are not row or collection) have a **Size** method, which returns the size of the type, and a **Variable** method, which indicates whether the instances of the type can be of variable size.

A row type might be transient. Row types have an array of **ITTypeInfo** references and strings that contain column type information and names. To obtain information from the columns in a row type, use the ColumnId(...) and ColumnType(...) methods.

Collection types expose the collection and the data type from which it is constructed. Collection types might have an upper limit on the number of elements. Collection types support the Size, Source, and Quality methods.

This class provides the following methods.

Method	Description
ITTypeInfo(const ITConnection &conn, const ITString &type_name, long size, long precision, long scale, long qualifier, ITBool byvalue, const MI_TYPEID *ptypeid = 0)	Constructs an ITTypeInfo object for an opaque data type.
ITTypeInfo(const ITConnection &conn, const ITString &type_name, const ITString &quality, const ITTypeInfo &memberType, const MI_TYPEID *ptypeid = 0)	Constructs an ITTypeInfo object for a collection data type.
ITTypeInfo(const ITConnection &conn, const ITString &type_name, const ITTypeInfo &source, const MI_TYPEID *ptypeid = 0)	Constructs an ITTypeInfo object for a distinct data type.
ITTypeInfo(const ITConnection &conn, const ITString &type_name, long ncols, ITTypeInfo **colps, const ITString *colnames, const MI_TYPEID *ptypeid = 0)	Constructs an ITTypeInfo object for a row data type.
ITTypeInfo(const ITConnection &conn, const ITString &type_name, const ITTypeInfo &consType, const ITTypeInfo &memberType, const MI_TYPEID *ptypeid = 0)	Constructs an ITTypeInfo object for a constructed data type.
ITTypeInfo(ITConnection &, const ITString &, long precision = -1, long scale = -1, long qualifier = -1)	Constructs an ITTypeInfo object with type information directly from the server. Other

Method	Description
	constructors get their type information about the client side without directly accessing the server.
ITTypeInfo *ConstructorType() const	Returns a pointer to an ITTypeInfo object that contains type information for the constructor object.
ITTypeInfo *MemberType() const	Returns a pointer to an ITTypeInfo object that contains type information for the member object of a collection or constructed type.
const ITString &Name() const	Returns the name of the database type.
ITBool IsSimple() const	Returns `TRUE` if this type is not a row type or a collection type.
ITBool IsRow() const	Returns `TRUE` if this type is a row type.
ITBool ByValue() const	Returns `TRUE` if the database type is passed by value, `FALSE` if it is passed by reference.
ITBool IsCollection() const	Returns `TRUE` if this type is a collection type.
ITBool IsConstructed() const	Returns `TRUE` if this type is a constructed type.
ITBool CompatibleType(const ITTypeInfo &) const	Returns `TRUE` if the argument is ITTypeInfo for the same type, a distinct type, a row type with the same column types, or a collection type with the same constructor and member type.
long Precision() const	Returns the precision (the number of significant digits) of a database type, if applicable.
long Qualifier() const	Returns the qualifier of the datetime or interval data type.
ITBool SameType(const ITTypeInfo &) const	Returns `TRUE` if the specified object is the same type as this object.
long Scale() const	Returns the scale of a database type, if applicable.
long Size() const	Returns `-1` if this is a variable-size type, or the size if the type is of fixed size.
long Bound() const	If the type is a variable-size type with a specified limit, this method returns the limit. For constructed types, the limit specifies the maximum number of items. Returns `-1` if no bound is in effect.

Method	Description
long Parameter() const	Returns the parameter of the type. For SQL numeric-derived types, returns the precision. For other numeric-derived types, returns the scale. For varchar-derived types, returns the maximum size.
ITBool Variable() const	Returns `TRUE` if the size is variable, or `FALSE` if the size is fixed.
ITBool IsDistinct() const	Returns `TRUE` if the type is distinct.
long ColumnCount() const	Returns the number of columns in this row type.
const ITString &ColumnName(long) const	Returns the name of the specified column.
long ColumnId(const ITString &) const	Returns the index of the given column name. Returns `-1` if the column name cannot be found.
const ITTypeInfo *ColumnType(long) const	Returns the type information of a column. Returns `NULL` if the column number is invalid.
const ITTypeInfo *ColumnType(const ITString &) const	Returns the type information of a column. Returns `NULL` if the column name cannot be found.
const ITTypeInfo *Source() const	Returns the type from which the current type was created as distinct. Returns `NULL` if the type does not have a source.
const ITString &Quality() const	Returns the collection type, such as `'SET'` or `'LIST'`.

Value interface reference

This section lists and describes the value interfaces. The ITFactoryList and ITPreserveData classes provide support for value interfaces.

The ITContainCvt interface

Base class: ITEssential

Decomposes an object into C++ base type instances. **ITContainCvt** is used by the **ITContainerIter** class to extract values from an object. **ITContainCvt** is to be used for objects that are naturally represented by base type arrays, such as a point list.

This interface provides the following methods.

Method	Description
ITBool ConvertTo(long item, output_type &)	Converts item to the output type. The output_type parameter must be one of the following types: short int long float double long double const char * ITString bool (if supported by compiler) ITInt8
long NumItems()	Returns the number of items in this object.
ITBool ConvertFrom (long item, const type)	Sets the value of the contained item from the value of the C++ type given as type.

The ITContainer interface

Base class: ITEssential

Returns one value from a set of values. **ITContainer** is used by the **ITContainerIter** class to iterate over the values contained in an object.

The *unknwn* argument of the GetItem method is the address of a pointer to an **ITEssential** interface of an object that will be the parent of any subobjects that might be created by the method. The newly created subobject returns its own **ITEssential** interface pointer in the argument if the object delegation was successful. The subobject reference count is 1 after the call, even if the **ITEssential** interface is passed back to the caller. The default argument is `NULL` to indicate that no object delegation is to be performed.

This interface provides the following methods.

Method	Description
long NumItems()	Returns the number of items in this object.
ITValue *GetItem(long position, ITEssential * *unknwn = NULL)	Returns the value interface pointer for a contained item. Returns NULL if the position is invalid.

The ITConversions interface

Base class: ITEssential

Interface to convert value objects to C++ base classes, strings, or value objects.

This interface provides the following methods.

Method	Description
ITBool ConvertTo(base_type &)	Converts to the variable of the specified type. Valid types for the base_type parameter are as follows: short int double long float long double const char * bool (if the C++ compiler supports it) ITString ITInt8
ITBool ConvertFrom(const type)	Sets the object from the value of the C++ type given as type.

C++ compiler interpretation of long doubles

HCL Informix® Object Interface for C++ provides data type conversion functions in the value interface ITConversions to enable conversion of C++ type long double. The intent is to permit fetching floating point values into C++ long double variables. However, the HCL Informix® Client Software Development Kit does not allow for conversion of long double values into HCL Informix® decimal or float types. Thus, Object Interface for C++ applications should always ensure that any floating literal passed to ITConversions::ConvertFrom(long double val) is within the double range. Otherwise, ConvertFrom(long double val) will return FALSE for value objects that contain SQL MONEY, FLOAT, and SMALLFLOAT values.

Object Interface for C++ is written with the assumption that a floating literal without the ANSI C++ specified suffixes l or L (example: 12.988 instead of 12.988L) assigned to a long double variable will be treated by the C++ compiler as a long double. This assumption agrees with the ANSI C++ Draft Standard (Doc No: X3J16/94-0027, WG21/N0414, 25 January 1994), which states that the type of a floating literal is double unless explicitly specified by a suffix. The suffixes f and F specify float; the suffixes l and L specify long double. Thus, the suffix l or L must be applied to a floating literal in order for it to be interpreted by the C++ compiler as a long double value.

Different versions of the Sun C++ compiler applied the ANSI C++ standard as it existed at the time of the compiler development and release. For example, Sun C++ 4.1 conforms to the ANSI standard described above, whereas pre-4.1 Sun C++ compilers always treated all floating literals, with or without the l and L suffixes, as long double values if they were

assigned to a long double variable. The following C++ code example demonstrates assignment of a floating literal to a long double variable, casting to a double, and comparison between the double and long double:

```
long double d = 12.988;
double dasd = (double) d;
if( dasd == d )
return 0;
else return 1;
```

The following table compares support for the ANSI C++ draft standard referenced above among several versions of Sun C ++ compilers. The table shows how the different compiler versions evaluated the expression (dasd == d). If the expression evaluates to FALSE, the values are not equal.

This interface provides the following methods.

Sun C++ compiler versions	**Evaluation of (dasd == d)**
4.0 (Dec 1993)	FALSE (values are not equal)
4.0.1 (Jul 1994)	FALSE (values are not equal)
4.1 (Oct 1995)	TRUE (values are equal)
5.0 (Oct 1999)	TRUE (values are equal)
6.01 (2001)	TRUE (values are equal)

The ITDateTime interface

Base class: ITValue

Allows access to the fields of a database time object (such as date, time, or interval).

This interface provides the following methods.

Method	**Description**
int Year()	Returns the year or years.
int Month()	Returns the month or months.
int Day()	Returns the day or days of the month.
int Hour()	Returns the hour or hours.
int Minute()	Returns the minute or minutes.
float Second()	Returns the second or seconds.
ITBool FromDate(int year, int month, int day)	Sets the date portions of the object exposing ITDateTime.
ITBool FromTime(int hour, int minute, float second)	Sets the time portions of the object exposing ITDateTime.

The ITDatum interface

Base class: ITValue

Provides access to the underlying data of a database class. It allows you to retrieve or set underlying data and determine their lengths. In addition, you can access the connection of the value object to the server.

All database classes that want to provide access to their underlying data expose this interface.

For some kinds of data (for example, row, collection, smartblob handle, character data) MI_DATUM is a pointer to the descriptor (MI_ROW *, MI_COLLECTION *, MI_LO_HANDLE *, MI_LVARCHAR *) rather than to the memory containing the data values. For these kinds of data **ITDatum::Data** returns a pointer to the descriptor. Pass a descriptor of the appropriate kind to SetData(). In addition, some of these descriptors are opaque (for example, MI_ROW). In these cases, the DataLength() return value is not usable and the data length SetData() argument is ignored.

This class provides the following methods.

Method	**Description**
MI_DATUM Data()	Returns an MI_DATUM encapsulated by the value object. Datum passing (by reference/value) obeys the same rules as mi_value() (see the *Informix® DataBlade® API Programmer's Guide* for information about mi_value()). If the datum is returned by reference, its memory is managed by the object. The application cannot modify the datum returned by reference.
long DataLength()	Returns the length of the datum encapsulated by the value object.
ITBool SetData (MI_DATUM data, long dataLen, ITPreserveData *preservedata = NULL)	Sets the value of a datum encapsulated by the value object to the parameter value. It returns `TRUE` if the operation was successful, `FALSE` otherwise.
const ITConnection & Connection()	Returns the connection of the value object.

The ITErrorInfo interface

Base class: ITEssential

The functionality provided by this class is only supported with HCL Informix® databases.

Extracts information about an error from an object. Some value objects, such as sets and large objects, can produce SQL errors, because SQL operations might be used to get the actual data values. If a value object can produce an SQL error, the value object supports the **ITErrorInfo** interface to enable the application to access the SQL error.

This interface provides the following methods.

Method	Description
ITBool Error()	Returns `TRUE` if an error occurred.
const ITString & SqlState()	Returns the ISO-standard SQL error code.
const ITString & ErrorText()	Returns the error message.

For an example of the use of this value object, see the `loex2.cpp` example application.

The ITEssential interface

ITEssential is the base class of the value interface classes. The **ITEssential** class is equivalent to the COM **IUnknown** interface of Microsoft™ and is abstract.

This interface provides the following methods.

Method	Description
ITOpErrorCode QueryInterface(const ITInterfaceID &ifiid, void **resultif)	Fills the parameter *resultif* with the address (or location) of the requested interface class. If the requested interface is not supported then the parameter is set to `NULL`.One of the following values is returned in ITOpErrorCode: IT_QUERYINTERFACE_FAILED-if the requested interface is not supported. IT_QUERYINTERFACE_SUCCESS-if the requested interface is successfully obtained. When the interface is no longer needed, it must be released by calling the Release() member function of **ITEssential**.
unsigned long AddRef()	Increments the reference count on this value object.
unsigned long Release()	Decrements the reference count on this value object. When the count reaches 0, the object might be freed, depending on the implementation of the value object.

The following definitions apply to the arguments and return values of the **ITEssential** interface and its descendants.

- *ITInterfaceID* is an index that identifies a particular value interface.
- *ITOpErrorCode* is a code returned from an interface method such as ITEssential::QueryInterface
- *ITOpErrorCode* indicates success or failure of a method. It is defined to be of the type long and can be assigned either the value `IT_QUERYINTERFACE_SUCCESS` or `IT_QUERYINTERFACE_FAILED`. The inline function ITIIDtoSID maps **ITInterfaceID**s to integral representations suitable for use in a **switch** statement.

By using the macros provided in the manner shown in the examples, value object implementors and application developers are protected from incompatibility with future versions of the interface.

Every interface defined by HCL Informix® has been given a unique interface identifier. These interface identifiers have an **IID** suffix, for example, **ITEssentialIID**.

The identifiers defined by the value interfaces are:

- ITContainCvtIID
- ITContainerIID
- ITConversionsIID
- ITDateTimeIID
- ITDatumIID
- ITErrorInfoIID
- ITEssentialIID
- ITLargeObjectIID
- ITRowIID
- ITSetIID
- ITValueIID

For details about the semantics of **ITEssential** when an object is delegated, see Object Containment and Delegation on page 44.

The ITLargeObject interface

Base class: ITEssential

Manipulates a large object returned by a query. Client value objects that are, in the server, based on large objects, expose an **ITLargeObject** interface; users creating such client value objects can use the **ITLargeObjectManager** class, which implements much of the functionality for accessing large objects.

This interface provides the following methods.

Method	Description
const MI_LO_HANDLE *Handle()	Returns the handle of the currently managed large object.
int Read(char *buf, int cnt)	Reads bytes from the large object at the current position.

Method	Description
int Write(const char *buf, int cnt)	Writes bytes to the large object at the current position.
ITInt8 Seek(ITInt8 off, int cntl = 0)	Sets the current position of the large object; *cntl* is a position like UNIX™ lseek (0 is absolute position, 1 is relative to current position, and 2 is relative to end of the large object).

ITBool SetHandle(const MI_LO_HANDLE *handle, int flags=MI_LO_RDWR)	Sets the specified DataBlade® API large object handle to this large object. The flags parameter is a bit mask argument with the following values:
	`MI_LO_RDONLY`
	`MI_LO_WRONLY`
	`MI_LO_RDWR`
	`MI_LO_TRUNC`
	`MI_LO_APPEND`
	`MI_LO_RANDOM`
	`MI_LO_SEQUENTIAL`
	`MI_LO_BUFFER`
	`MI_LO_NOBUFFER`

ITInt8 Size()	Returns the total size of the large object.
ITBool SetSize(ITInt8)	Sets the total size of the large object.

The ITRow interface

Base class: ITValue

Provides access to row values. A row value can extract references to the number of columns it contains and the value of a specific column.

The *unknwn* argument of the **Column** method is the address of a pointer to an **ITEssential** interface of an object that will be the parent of any subobjects created by the method. The newly created subobject returns its own **ITEssential** interface pointer in the argument if the object delegation was successful. The subobject reference count is 1 after the call, even if the **ITEssential** interface is passed back to the caller. The default argument is `NULL` to indicate that no object delegation is to be performed.

This interface provides the following methods.

Method	Description
long NumColumns()	Returns the number of columns in this row value.
ITValue *Column(long, ITEssential **unknwn = NULL)	Returns a pointer to the value interface of a column.

Method	Description
ITValue *Column(const ITString &, ITEssential **unknwn = NULL)	Returns a pointer to the value interface of a column by name. Returns `NULL` if you specify an invalid column name.

The ITSet interface

Base class: ITValue

The **ITSet** class provides random access to rowset or collection members.

This interface provides the following methods.

Method	Description
ITBool IsScrollable()	Returns `TRUE` if this set is scrollable.
ITBool IsReadOnly()	Returns `TRUE` if this set cannot be updated.
ITBool Open()	Opens or reopens the set.
ITBool Close()	Closes the set. Close does not release the interface.
ITBool Delete(enum ITPosition pos = ITPositionCurrent, long jump = 0) = 0	Deletes the specified member from the set. Returns `TRUE` if successful, `FALSE` otherwise.
ITBool Insert(ITDatum *item, enum ITPosition pos = ITPositionCurrent, long jump = 0) = 0	Inserts the specified item immediately after the current item. Returns `TRUE` if successful, `FALSE` otherwise.
ITValue *MakeItem(ITEssential **outerunkn = NULL)	Returns a pointer to an ITValue interface of a new object of the same type as the objects in the collection. The value of the object can then be set (for example, with FromPrintable()) and the object can be inserted into the collection object.
ITValue *Fetch(ITEssential **outerunkn = NULL, enum ITPosition pos = ITPositionNext, long jump = 0)	Fetches the collection member and returns the pointer to its ITValue interface.

The ITValue interface

Base class: ITEssential

An interface class that provides basic value object support. All objects representing values from the database must support, at a minimum, the **ITValue** interface.

This interface provides the following methods.

Method	Description
const ITString &Printable()	Returns a printable form of the value in a constant string.
const ITTypeInfo &TypeOf()	Returns the database type information for this value.
ITBool IsNull()	Returns `TRUE` if this is a null value.
ITBool SameType(ITValue *)	Returns `TRUE` if this value is the same database type as the specified value.
ITBool Equal(ITValue *)	Returns `TRUE` if the specified values are equal. False values are not equal to each other or to any other value.
ITBool LessThan(ITValue *)	Returns `TRUE` if and only if the object is less than the argument and the objects are comparable. ("Less than" is defined as appropriate for the data type.)
ITBool CompatibleType(ITValue *)	Returns `TRUE` for all built-in objects if the objects are compatible.
ITBool IsUpdated()	Returns `TRUE` if the object was updated, `FALSE` if it did not change since it was first created. Value objects of complex types (rows, collections) are considered updated when any of their members are updated.
ITBool FromPrintable(const ITString &printable)	Sets the object value from the printable representation. FromPrintable() accepts the printable representation of the object equivalent to the input function of the object. Printable() provides the character representation of the object equivalent to the output function of the object. For additional usage, see Use of ITValue::Printable with null value objects on page 88.
ITBool SetNull()	Sets the object value to `NULL`.

Use of ITValue::Printable with null value objects

value objects can encapsulate a datum fetched from a database or a datum that is to be inserted into a database. A value object exists only in the client application, and the datum encapsulated by it can be written to the database by using prepared statements encapsulated by ITStatement objects or, if a cursor that can be updated is used, by ITCursor::UpdateCurrent.

After it fetches a row from a database in which there are columns containing SQL NULL entries (that is, with no data), ITValue::Printable called on a value object matching a NULL column will return the string "null." The string "null" is informational only.

Likewise, after ITValue::SetNull is called to set a value object to null (where the term "null" means SQL NULL: That is, no data), calls to ITValue::Printable return the string "null" for that value object to indicate that the value object contains no data.

In the special case where the Object Interface for C++ application program inserted the valid data string "null" into a value object (for example, by calling ITValue::FromPrintable("null") or by fetching it from a database), the application can still distinguish between a null value object and a value object containing the valid data "null" by calling the function ITValue::IsNull on the value object. ITValue::IsNull returns true if the value object is null and false if the value object contains the valid data "null." Calling ITValue::IsNull is the preferred way to determine if a value object contains no data and is to be used instead of ITValue::Printable.

Appendixes

This section contains additional reference information.

Supported data types

This section lists the server data types and the interfaces supported for them.

Tip: Objects that are BLOB and CLOB objects implemented as part of the library return the textual value of the smart large object handle through the ITValue::Printable method and set it through ITValue::FromPrintable.

Simple large objects (TEXT and BYTE types) are represented on the client as data in RAM. Use the offset operator [] in queries to limit the amount of data retrieved by the client. To update a simple large object in the server, pass the value object that encapsulates the simple large object data as a prepared statement parameter.

Table 1. Data types and supported interfaces

Data type	ITEssential interface	ITValue interface	ITRow interface	ITConversions interface	ITLargeObject interface	ITSet interface
blob*	X	X			X	
boolean*	X	X		X		
byte	X	X				
char	X	X		X		
character	X	X		X		
char1	X	X		X		
cXob*	X	X			X	
date	X	X				
datetime	X	X				
decimal	X	X		X		

Table 1. Data types and supported interfaces

(continued)

Data type	ITEssential interface	ITValue interface	ITRow interface	ITConversions interface	ITLargeObject interface	ITSet interface
double precision	X	X		X		
int8*	X	X		X		
integer	X	X		X		
interval day to second	X	X				
interval year to month	X	X				
money	X	X		X		
numeric	X	X		X		
real	X	X		X		
smallint	X	X		X		
text	X	X				
collection* **	X	X				X
row* **	X	X	X			
ITQuery::ExecToSet result set **	X	X				X

* Supported only by HCL Informix®

** Constructed type

Table 2. More supported interfaces

Server base type	ITDateTime	ITContainer	ITErrorInfo	ITContainCvt	ITDatum
blob*			X		X
boolean*					X
byte					X
char					X
character					X

Table 2. More supported interfaces

(continued)

Server base type	ITDateTime	ITContainer	ITErrorInfo	ITContainCvt	ITDatum
char1					X
cXob*			X		X
date	X				X
datetime	X				X
decimal					X
double precision					X
int8*					X
integer					X
interval day to second	X				X
interval year to month	X				X
money					X
numeric					X
real					X
smallint					X
text					X
collection* **					X
row* **		X			X
ITQuery::ExecToSet result set **					

* Supported only by HCL Informix®

** Constructed type

Example programs

For the path and name of the directory containing the example files, consult the latest release notes. The examples directory also contains a makefile to build the examples.

The following is a list of the example programs with a brief description of each:

- `cnvex.cpp` uses the **ITConversions** interface to convert an **integer** to other types.
- `contain.cpp` shows how containers are used with the **ITContainer** interface (only).
- `csql.cpp` is a simple query example.
- `csql2.cpp` is a simple query example that uses error callbacks.
- `csql3.cpp` is a simple query example monitoring the transaction state of the connection.
- `curstst.cpp` opens a cursor and scrolls through the result set in various ways.
- `cursupd.cpp` illustrates the use of a cursor with parameter markers to update the database.
- `delegate.cpp` is an example of object delegation.
- `dtex.cpp` is a date/time interface example.
- `fsexamp1.cpp` illustrates iteration through a container.
- `ifval.cpp` is an example of a value object supporting multiple interfaces.
- `loadtab.cpp` loads a table from a text file by using a prepared statement with **ITStatement** (HCL Informix® only).
- `loex1.cpp` illustrates access to a large object through the **ITLargeObject** interface (HCL Informix® only).
- `loex2.cpp` is a large object and error information example. (HCL Informix® only).
- `queryex.cpp` illustrates the use of transaction control within a query.
- `rawval.cpp` illustrates access to the **ITDatum** interface of a large object.
- `rowref.cpp` is an example of a value object with multiple interfaces and copy-on-update.
- `rowset.cpp` retrieves rows into a set.
- `simpval.cpp` is an example of a value object derived from **ITDatum**.
- `tabcnt.cpp` is a simple example issuing a query and uses value interfaces.
- `testtype.cpp` is an example of a dynamically loaded value object.

The ITLocale class

This section describes the **ITLocale** class, which encapsulates the GLS API.

ITLocale methods perform:

- Locale-sensitive conversions between the text and binary forms of the date, time, numeric, and money data types
- General string and character manipulation, such as comparison and classification, for multibyte and wide character strings and characters

Multibyte character string termination

Some APIs that use **ITLocale** assume that character strings are terminated with a null character, while others assume that a string consists of a pointer and length indicating the number of bytes in the string. **ITLocale** methods can be used in both cases.

Multibyte character strings are passed to **ITLocale** methods in two arguments:

- `const char *s` specifies a multibyte string.
- `int nbytes` specifies the length of the string.

The actual argument names might vary.

If *nbytes* is the value `ITLocale::ScanToNul`, the method treats *s* as a null-terminated string. Otherwise, the method assumes *s* contains *nbytes* bytes.

The terminator of a null-terminated string is a single byte whose value is `0`. Multibyte character strings that are not null-terminated might contain null characters, but these characters do not indicate the end of the string.

Multibyte character termination

A multibyte character passed to an **ITLocale** method is represented with two arguments:

- `const char *mchar` specifies a multibyte character.
- `int nmcharbytes` specifies the number of bytes that represent the multibyte character.

The actual argument names might vary.

If *nmcharbytes* is `ITLocale::ScanNoLimit`, the method reads bytes at *mchar* until a complete character is formed. Otherwise it reads no more than *nmcharbytes* bytes at *mchar* to form a character.

Memory allocation

The GLS API performs no memory allocation or deallocation. Therefore, you must allocate an appropriately sized buffer for any **ITLocale** method that returns a string. You must also deallocate the memory for the buffer when the method is through with it.

Access the ITLocale object

An application has a single **ITLocale** object. The ITLocale::Current() method returns a pointer to the object. The constructor that creates the **ITLocale** object is protected and cannot be called directly.

Error return method

This method returns a GLS error number.

```
int GetError() const
```

Some **ITLocale** methods indicate whether an error has occurred in their return values (`-1`, `0`, or `NULL`). For other methods, you must call ITLocale::GetError() to determine if there was an error. You can, as a standard practice, call ITLocale::GetError() after every call to an **ITLocale** method.

See the description of the corresponding function in the *Informix® GLS User's Guide* to see the errors that a particular **ITLocale** method can return.

String comparison methods

This section describes the **ITLocale** methods for comparing strings.

The MCollate method

This method compares multibyte character string *s1* to multibyte character string *s2* for sort order according to the rules of the current locale.

```
int MCollate(const char *s1, const char *s2,
   int nbytes1 = ITLocale::ScanToNul,
   int nbytes2 = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns an integer that is:

- Greater than 0 if *s1* is greater than (after) *s2* in sort order
- Less than 0 if *s1* is less than (before) *s2* in sort order
- 0 if *s1* is equal to *s2* in sort order

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WCollate method

This method compares wide character string *s1* to wide character string *s2* for sort order according to the rules of the current locale.

```
int WCollate(const ITWChar *s1, const ITWChar *s2,
   int nwchars1 = ITLocale::ScanToNul,
   int nwchars2 = ITLocale::ScanToNul) const
```

The *nwchars1* and *nwchars2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns an integer that is:

- Greater than 0 if *s1* is greater than (after) *s2* in sort order
- Less than 0 if *s1* is less than (before) *s2* in sort order
- 0 if *s1* is equal to *s2* in sort order

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

String processing methods

This section describes the **ITLocale** methods for processing strings.

The MConcatenate method

This method appends multibyte character string *s2* to the end of multibyte character string *s1*. If the two strings overlap, the results are undefined.

```
int MConcatenate(char *s1, const char *s2,
   int nbytes1 = ITLocale::ScanToNul,
   int nbytes2 = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns the length in bytes of the resulting concatenated string, not including the null terminator if there is one.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MScan method

This method searches for the first occurrence of the multibyte character *mchar* in the multibyte character string *s*.

```
char *MScan(const char *s, const char *mchar,
   int nstrbytes = ITLocale::ScanToNul,
   int nmcharbytes = ITLocale::ScanNoLimit) const
```

The *nstrbytes* parameter specifies the length of the corresponding string *s*. You can provide an integer to specify the number of bytes in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

The *nmcharbytes* parameter specifies the length of the corresponding multibyte character *mchar.* You can provide an integer to specify the number of bytes in *mchar*, in which case this method reads up to this many bytes from *mchar* when trying to

form a complete character. Or you can set *nmcharbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns a pointer to the first occurrence of the multibyte character *mchar* in the string *s*. If *mchar* is not found in *s*, this method returns `NULL`. If you call ITLocale::GetError(), it returns `0`.

Related reference

Error return method on page 93

The MCopy method

This method copies the multibyte character string *from* to the location pointed to by *to*. If *from* and *to* overlap, the results of the method are undefined.

```
int MCopy(char *to, const char *from,
      int nfrombytes = ITLocale::ScanToNul) const
```

The *nfrombytes* parameter specifies the length of the corresponding string *from*. You can provide an integer to specify the number of bytes in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *from* is a null-terminated string.

This method returns the number of bytes in the resulting string, not including the null terminator, if there is one.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MComplSpanSize method

This method returns the number of characters in the longest initial substring of multibyte character string *s1* that consists entirely of multibyte characters not in the string *s2*.

```
int MComplSpanSize(const char *s1, const char *s2,
            int nbytes1 = ITLocale::ScanToNul,
            int nbytes2 = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MLength method

This method returns the number of characters (not bytes) in the multibyte character string *s*, not including the null terminator, if there is one.

```
int MLength(const char *s, int nbytes =
          ITLocale::ScanToNul) const
```

The *nstrbytes* parameter specifies the length in bytes of the corresponding string s. You can provide an integer to specify the number of bytes in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MFindSubstr method

This method searches for the first occurrence of the multibyte string *s2* in the multibyte string *s1*.

```
char *MFindSubstr(const char *s1, const char *s2,
              int nbytes1 = ITLocale::ScanToNul,
              int nbytes2  = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length in bytes of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns a pointer to the first occurrence of the multibyte string *s2* in *s1*.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MNConcatenate method

This method appends one or more multibyte characters in the *from* multibyte string to the end of the multibyte string *to*. If *from* and *to* overlap, the results of this method are undefined.

```
int MNConcatenate(char *to, const char *from, int limit,
            int ntobytes = ITLocale::ScanToNul,
            int nfrombytes = ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of characters to read from the *from* string.

The *ntobytes* and *nfrombytes* parameters specify the length of the *to* and *from* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns the number of bytes in the resulting string.

If there is an error, the method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MNCopy method

This method copies the specified number of multibyte characters in *from* to the location pointed to by *to*.

```
int MNCopy(char *to, const char *from, int limit,
         int nfrombytes = ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of characters to read from the *from* string.

The *nfrombytes* argument specifies the length of the corresponding string *from*. You can provide an integer to specify the number of bytes in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *from* is a null-terminated string.

This method returns the length in bytes of the resulting copied string, not including the null terminator if there is one.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MNTSBytes method

This method returns the number of bytes in the multibyte character string *s*, not including any trailing space characters. The characters not included in the count are the ASCII space character and any multibyte characters equivalent to the ASCII space character.

```
int MNTSBytes(const char *s, int nbytes = ITLocale::ScanToNul) const
```

Space characters embedded in the string before the series of spaces at the end of the string are included in the count.

The *nbytes* parameter specifies the length of the corresponding string *s*. You can provide an integer to specify the number of bytes in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MNTSLength method

This method returns the number of characters in the multibyte character string *s*, not including any trailing space characters.

```
int MNTSLength(const char *s, int nbytes = ITLocale::ScanToNul) const
```

The characters not included in the count are the ASCII space character and any multibyte characters equivalent to the ASCII space character. Space characters embedded in the string before the series of spaces at the end of the string are included in the count.

The *nbytes* parameter specifies the length of the corresponding string *s*. You can provide an integer to specify the number of bytes in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MSpan method

This method searches for the first occurrence in the multibyte character string *s1* of any character from the multibyte character string *s2*.

```
char *MSpan(const char *s1, const char *s2,
          int nbytes1 = ITLocale::ScanToNul,
          int nbytes2 = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns a pointer to the first occurrence in *s1* of any character from *s2*. If no character from *s2* is found in *s1* the method returns `NULL` and ITLocale::GetError() returns `0`.

If an error occurs, the method returns `NULL` and ITLocale::GetError() returns a specific error message.

Related reference

Error return method on page 93

The MRScan method

This method locates the last occurrence of multibyte character *c* in the multibyte character string *s*.

```
char *MRScan(const char *s, const char *c,
           int nsbytes = ITLocale::ScanToNul,
           int ncbytes = ITLocale::ScanNoLimit) const
```

The *nsbytes* parameter specifies the length of the corresponding string *s*. You can provide an integer to specify the number of bytes in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

The *ncbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *ncbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns a pointer to the last occurrence of the multibyte character *c* in the string *s*. If this method does not find *c* in *s*, it returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MSpanSize method

This method returns the number of characters in the longest initial substring of multibyte character string *s1* that consists entirely of multibyte characters in the string *s2*.

```
int MSpanSize(const char *s1, const char *s2,
            int nbytes1 = ITLocale::ScanToNul,
            int nbytes2 = ITLocale::ScanToNul) const
```

The *nbytes1* and *nbytes2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of bytes in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WConcatenate method

This method appends a copy of the wide character string *from* to the end of the wide character string *to*. If *from* and *to* overlap, the results of this method are undefined.

```
int WConcatenate(ITWChar *to, const ITWChar *from,
              int nfromwchars = ITLocale::ScanToNul,
              int ntowchars = ITLocale::ScanToNul) const
```

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WScan method

This method locates the first occurrence of wide character *c* in the wide character string *s*.

```
ITWChar *WScan(const ITWChar *s, ITWChar c,
               int nswchars = ITLocale::ScanToNul) const
```

The *nswchars* parameter specifies the length of the corresponding wide character string *s*. You can provide an integer to specify the number of characters in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

This method returns a pointer to the first occurrence of *c* in *s*. If this method does not find *c* in *s*, it returns `NULL` and ITLocale::GetError() returns `0`.

If there is an error, this method returns `NULL` and ITLocale::GetError() returns a specific error number.

Related reference

Error return method on page 93

The WCopy method

This method copies the wide character string *from* to the location pointed to by *to*. If the strings overlap, the result is undefined.

```
int WCopy(ITWChar *to, const ITWChar *from,
          int nfromwchars = ITLocale::ScanToNul) const
```

The *nfromchars* parameter specifies the length in characters of the corresponding wide character string *from*. You can provide an integer to specify the number of characters in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

This method returns the number of characters in the resulting string, not including the null terminator, if there is one.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WComplSpanSize method

```
int WComplSpanSize(const ITWChar *s1, const ITWChar *s2,
            int nwchars1 = ITLocale::ScanToNul,
            int nwchars2 = ITLocale::ScanToNul) const
```

This method returns the number of wide characters in the maximum initial substring of the wide character string *s1* that consists entirely of wide characters not in the wide character string *s2*.

The *nwchars1* and *nwchars2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WLength method

This method returns the number of wide characters in the wide character string *s*, not including the null terminator, if there is one. No errors are defined for this method.

Example

```
int WLength(const ITWChar *s) const
```

The WNConcatenate method

This method appends wide character string *from* to the end of wide character string *to*. If *from* and *to* overlap, the results of this method are undefined.

```
int WNConcatenate(ITWChar *to, const ITWChar *from,
            int limit,
            int nfromwchars = ITLocale::ScanToNul,
            int ntowchars = ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of characters to read from the *from* string and append to the *to* string.

The *ntowchars* and *nfromwchars* parameters specify the length of the *to* and *from* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns the number of wide characters in the resulting concatenated string, not including the null terminator, if there is one.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WNCopy method

This method copies wide character string *from* to the location pointed to by *to*. If *from* and *to* overlap, the results of this method are undefined.

```
int WNCopy(ITWChar *to, const ITWChar *from, int limit,
           int nfromwchars = ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of characters to read from the *from* string and append to the *to* string.

The *nfromwchars* parameter specifies the length of the corresponding wide character string *from*. You can provide an integer to specify the number of characters in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *from* is a null-terminated string.

This method returns the number of wide characters copied.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WNTSLength method

This method returns the number of characters in the wide character string *s*, not including any trailing space characters.

```
int WNTSLength(const ITWChar *s, int nwchars = ITLocale::ScanToNul) const
```

The characters not included in the count are the ASCII space character and any wide characters equivalent to the ASCII space character.

The *nwchars* parameter specifies the length of the corresponding wide character string *s*. You can provide an integer to specify the number of characters in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WSpan method

This method searches for the first occurrence in the wide character string *s1* of any wide character from the string *s2*.

```
ITWChar *WSpan(const ITWChar *s1, const ITWChar *s2,
            int nwchars1 = ITLocale::ScanToNul,
            int nwchars2 = ITLocale::ScanToNul) const
```

The *nwchars1* and *nwchars2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns a pointer to the first occurrence of the wide character string *s1* in the string *s2*. If this method does not find *s1* in *s2*, it returns `NULL`. If you call ITLocale::GetError(), it returns `0`.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WRScan method

This method locates the last occurrence of wide character *c* in the wide character string *s*.

```
ITWChar *WRScan(const ITWChar *s, ITWChar c,
             int nswchars = ITLocale::ScanToNul) const
```

The *nswchars* parameter specifies the length of the corresponding wide character string *s*. You can provide an integer to specify the number of characters in *s*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *s* is a null-terminated string.

This method returns a pointer to the last occurrence of wide character *c* in wide character string *s*. If this method does not find *c* in *s*, it returns `NULL`. If you call ITLocale::GetError(), it returns `0`.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WSpanSize method

This method returns the number of characters in the longest initial substring of the wide character string *s1* that consists entirely of characters from the wide character string *s2*.

```
int WSpanSize(const ITWChar *s1, const ITWChar *s2,
            int nwchars1 = ITLocale::ScanToNul,
            int nwchars2 = ITLocale::ScanToNul) const
```

The *nwchars1* and *nwchars2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WFindSubstr method

This method searches for the first occurrence of the wide character string *s2* in the wide character string *s1*.

```
ITWChar *WFindSubstr(const ITWChar *s1, const ITWChar *s2,
            int nwchars1 = ITLocale::ScanToNul,
            int nwchars2 = ITLocale::ScanToNul) const
```

The *nwchars1* and *nwchars2* parameters specify the length of the *s1* and *s2* strings. You can provide an integer to specify the number of characters in the corresponding string. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that the corresponding string is null-terminated.

This method returns a pointer to the first occurrence of the wide character string *s1* in wide character string *s2*. If this method does not find *s1* in *s2*, it returns `NULL`. If you call ITLocale::GetError(), it returns `0`.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Environment method

This topic describes the **ITLocale** method for determining the client locale.

```
const char *LocaleName() const
```

This method returns the value of the GLS environment variable **CLIENT_LOCALE**.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Code set conversion methods

This section describes the **ITLocale** methods for converting code sets.

The ConvertCodeset method

This method converts the string of multibyte characters in *from* to another code set and copies the result to the location pointed to by *to*.

Important: This method assumes that *from* points to a null-terminated string.

```
int ConvertCodeset(char *to, const char *from,
                   const char *toLocaleName,
                   const char *fromLocaleName) const
```

Use the *fromLocalName* parameter to identify the locale from which you are converting. Use the *toLocalName* parameter to specify the locale to which you are converting. There is a single code set associated with each locale. By identifying the locale, you also specify the code set.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

However, there are significant differences between the parameters of the GLS API function ifx_gl_cv_mconv() and the **ITLocale** method. For example, the GLS API function has specific code set parameters, whereas ITLocale::ConvertCodeset has locale name parameters that imply the code set name. Also, the GLS API function has additional parameters for copying fragments of strings that are unavailable to **ConvertCodeset**.

Related reference

Error return method on page 93

The NeedToConvertCodeset method

This method determines if conversion is necessary from the code set associated with the *fromLocaleName* locale to the code set associated with the *toLocalename* locale.

```
int NeedToConvertCodeset(const char *toLocaleName,
                         const char *fromLocaleName) const
```

This method returns `1` if conversion is needed and `0` if not.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The SizeForCodesetConversion method

This method calculates the number of bytes needed to convert the multibyte characters in *nfrombytes* from the code set associated with the *fromLocaleName* locale to the code set associated with the *toLocaleName* locale.

```
int SizeForCodesetConversion(const char *toLocaleName,
                             const char *fromLocaleName,
                            int nfrombytes) const
```

This method returns the number of bytes to convert. If this value equals the number of bytes in *nfrombytes*, then conversion is done in place. Otherwise, you must allocate another buffer for the conversion.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Character classification methods

This section describes the **ITLocale** methods for classifying characters.

The IsAlnum method

This method determines whether a multibyte character *c* or a wide character *c* is an alphanumeric character according to the rules of the current locale.

```
ITBool IsAlnum(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsAlnum(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is an alphanumeric character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsAlpha method

This method determines whether multibyte character *c* or wide character *c* is an alphabetic character according to the rules of the current locale.

```
ITBool IsAlpha(const char *c, int nbytes = ITLocale::ScanNoLimitl) const

ITBool IsAlpha(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is an alphabetic character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsBlank method

This method determines whether multibyte character *c* or wide character *c* is a blank character (space or tab, single or multibyte), according to the rules of the current locale. Blank characters include the single-byte space and tab characters and any multibyte version of these characters.

```
ITBool IsBlank(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsBlank(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a blank or tab; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsCntrl method

This method determines whether multibyte character *c* or wide character *c* is a control character according to the rules of the current locale.

```
ITBool IsCntrl(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsCntrl(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a control character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

For more information, see the **ifx_gl_ismcntrl** function in the *Informix® GLS User's Guide*.

Related reference

Error return method on page 93

The IsDigit method

This method determines whether multibyte character *c* or wide character *c* is a digit character according to the rules of the current locale.

```
ITBool IsDigit(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsDigit(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a digit character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsGraph method

This method determines whether multibyte character *c* or wide character *c* is a graphical character according to the rules of the current locale.

```
ITBool IsGraph(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsGraph(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a graphical character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsLower method

This method determines whether multibyte character *c* or wide character *c* is a lowercase character according to the rules of the current locale.

```
ITBool IsLower(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsLower(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a lowercase character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsPrint method

This method determines whether multibyte character *c* or wide character *c* is a printable character according to the rules of the current locale. Printable characters include all characters except control characters.

```
ITBool IsPrint(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsPrint(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a printable character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsPunct method

This method determines whether multibyte character *c* or wide character *c* is a punctuation character according to the rules of the current locale. Punctuation characters include any single-byte ASCII punctuation characters and any non-ASCII punctuation characters.

```
ITBool IsPunct(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsPunct(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a printable character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsSpace method

This method determines whether multibyte character *c* or wide character *c* is a space character according to the rules of the current locale. Space characters include the blank characters (blank and tab) as well as the single-byte and multibyte versions of the newline, vertical tab, form-feed, and carriage return characters.

```
ITBool IsSpace(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsSpace(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a space character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsUpper method

This method determines whether multibyte character *c* or wide character *c* is an uppercase character according to the rules of the current locale.

```
ITBool IsUpper(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsUpper(ITWChar c) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *c*. You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is an uppercase character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The IsXDigit method

This method determines whether multibyte character *c* or wide character *c* is a hexadecimal number character according to the rules of the current locale.

```
ITBool IsXDigit(const char *c, int nbytes = ITLocale::ScanNoLimit) const

ITBool IsXDigit(ITWChar c) const
```

Only the ten ASCII digit characters are in the hexadecimal class. Multibyte versions of these digits or alternative representations of these digits (for example, Hindi or Kanji digits) are not in this class, but instead are in the alpha class.

The *nbytes* parameter specifies the length of the corresponding multibyte character *c* You can provide an integer to specify the number of bytes in *c*, in which case this method reads up to this many bytes from *c* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns `TRUE` if *c* is a hexadecimal number character; otherwise it returns `FALSE`.

If there is an error, this method returns `FALSE`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Character case conversion methods

This section describes the **ITLocale** methods for converting the case of characters.

The ToUpper—Wide Character method

This method converts the wide character *c* to uppercase. If the wide character has no uppercase equivalent, it is copied unchanged.

```
ITWChar ToUpper(ITWChar c) const
```

This method returns the uppercase character equivalent, if there is one, or the input character if there is no uppercase equivalent. If there is an error, this method returns `0`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ToUpper—Multibyte Character method

This method converts the multibyte characters in *from* to uppercase. If the characters in *from* have no uppercase equivalent, they are copied unchanged.

```
unsigned short ToUpper(char *to,
               const char *from,
               unsigned short &nfrombytes,
               int nbytes = ITLocale::ScanNoLimit)const
```

This method returns in the *nfrombytes* parameter the number of bytes read from the location pointed to by *from*. You must pass the address of an **unsigned short** for this parameter.

The *nbytes* parameter specifies the length of the multibyte characters in *from*. You can provide an integer to specify the number of bytes, in which case this method reads up to this many bytes from *from* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns the number of bytes actually copied to the buffer pointed to by *to*.

If there is an error, this method returns `0`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ToLower—Wide Character method

This method converts the wide character *c* to lowercase.

```
ITWChar ToLower(ITWChar c) const
```

This method returns the lowercase equivalent of the input character. If there is no lowercase equivalent, the method returns the input character. If there is an error, this method returns `0`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ToLower—Multibyte Character method

This method converts the multibyte characters in *from* to lowercase.

```
unsigned short ToLower(char *to, const char *from,
            unsigned short &nfrombytes,
            int nbytes = ITLocale::ScanNoLimitl) const
```

The *nfrombytes* parameter specifies the number of bytes to copy.

The *nbytes* parameter specifies the length of the multibyte characters in *from*. You can provide an integer to specify the number of bytes, in which case this method reads up to this many bytes from *from* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

If there is an error, this method returns `0`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Built-in data type conversion methods

This section describes the **ITLocale** methods for converting built-in data types to an internal representation.

The ConvertDate method

This method converts the date pointed to by *str* into an internal representation.

```
mi_date ConvertDate(const ITString &str,
         const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the internal representation. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns the internal representation of the date.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The FormatDate method

This method creates a date string from the **mi_date** structure pointed to by *d*.

```
ITString FormatDate(const mi_date *d,
          const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the date string. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns a date string.

If there is an error, this method returns an empty **ITString** object. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ConvertDatetime method

This method converts the date-time string pointed to by *str* into an internal representation.

```
mi_datetime ConvertDatetime(const ITString &str,
         const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the internal representation. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns the internal representation of the date.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The FormatDatetime method

This method creates a date-time string from the **mi_datetime** structure pointed to by *dt*.

```
ITString FormatDatetime(const mi_datetime *dt,
      const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the date string. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns a date-time string.

If there is an error, this method returns an empty **ITString** object. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ConvertNumber method

This method converts the number string pointed to by *str* into an internal representation.

```
mi_decimal ConvertNumber(const ITString &str,
      const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the internal representation. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns the internal representation of the number.

If there is an error, this method returns a null **mi_decimal** value. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The FormatNumber method

This method creates a number string from the **mi_decimal** structure pointed to by *dec*.

```
ITString FormatNumber(const mi_decimal *dec,
      const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the date string. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns a number string.

If there is an error, this method returns an empty **ITString** object. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The ConvertMoney method

This method converts the money string pointed to by *str* into an internal representation.

```
mi_money ConvertMoney(const ITString &str,
         const ITString &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the internal representation. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation..

This method returns the internal representation of the money string in an **mi_money** structure.

If there is an error, this method returns a null **mi_money** value. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The FormatMoney method

This method creates a money string from the **mi_money** structure pointed to by *m*.

```
ITString FormatMoney(const mi_money *m,
                const ITString
                &format = ITString::Null) const
```

Use the *format* parameter to specify the format of the date string. If you set format to `NULL` (the default), the format is determined by the environment.

If you do not specify `NULL` for the format, you must pass a string to *format* defining the format of the internal representation.

This method returns a number string.

If there is an error, this method returns an empty **ITString** object. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Multibyte and wide character conversion methods

This section describes the **ITLocale** methods for converting characters and character strings between their multibyte and wide character representations.

The MToWString method

This method converts the multibyte character string *from* to its wide character representation and stores the result in *to*.

```
int MToWString(ITWChar *to, const char *from, int limit,
            int nfrombytes = ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of bytes to read from the *from* string and write to *to*.

The *nfrombytes* parameter specifies the length of the corresponding multibyte string *from*. You can provide an integer to specify the number of bytes in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *from* is a null-terminated string.

This method returns number of characters read from *from* and written to *to*, not counting the null terminator, if there is one.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MToWChar method

This method converts the multibyte character *from* into its wide character representation.

```
ITWChar MToWChar(const char *from, int nfrombytes = ITLocale::ScanNoLimit) const
```

The *nfrombytes* parameter specifies the length of the corresponding multibyte character *from*. You can provide an integer to specify the number of bytes in *from* in which case this method reads up to this many bytes from *from* when trying to form a complete character. Or you can set *nfrombytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

This method returns the wide character representation of multibyte character *from*.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

The GLS API function ifx_gl_mbtowc() has different parameters from MToChar. The GLS API function returns the wide character in the parameter list and returns the number of bytes read in the function return value.

Related reference

Error return method on page 93

The WToMString method

This method converts the wide character string *from* to its multibyte representation and stores it in the location pointed to by *to*.

```
int WToMString(char *to, const ITWChar *from, int limit,
            int nfromsize =
            ITLocale::ScanToNul) const
```

Use *limit* to specify the maximum number of bytes to read from the *from* string and write to *to*. If a character to be written to *to* would cause more than the specified limit of bytes to be written, no part of that character is written. In this case the method writes less than the specified limit of bytes.

The *nfromsize* parameter specifies the length of the corresponding string *from*. You can provide an integer to specify the number of bytes in *from*. Or you can use the constant `ITLocale::ScanToNul` (the default) to specify that *from* is a null-terminated string.

This method returns the number of bytes it writes to multibyte string *to*.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The WToMChar method

This method converts the wide character *from* to its multibyte representation and stores it in consecutive bytes starting at the location pointed to by *to*.

```
int WToMChar(char *to, const ITWChar from) const
```

This method returns the number of bytes it writes to *to*.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Multibyte string traversal and indexing methods

This section describes the following **ITLocale** methods for converting built-in data types to an internal representation.

The MCharBytes method

This method returns the maximum number of bytes that any multibyte character can occupy.

Example

```
int MCharBytes() const
```

The MCharLen method

This method returns the number of bytes in the multibyte character *s*.

```
int MCharLen(const char *s, int nbytes = ITLocale::ScanToNul) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *s*. You can provide an integer to specify the number of bytes in *s*, in which case this method reads up to this many bytes from *s* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

If there is an error, this method returns `-1`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MNextChar method

This method returns a pointer to the next multibyte character after the multibyte character *s*.

```
char *MNextChar(const char *s, int nbytes = ITLocale::ScanNoLimit) const
```

The *nbytes* parameter specifies the length of the corresponding multibyte character *s*. You can provide an integer to specify the number of bytes in *s*, in which case this method reads up to this many bytes from *s* when trying to form a complete character. Or you can set *nbytes* to `ITLocale::ScanNoLimit` (the default), in which case this method reads as many bytes as necessary to form a complete character.

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

The MPrevChar method

This method returns a pointer to the first byte of the multibyte character before the multibyte character *c*, where *s* is a pointer to the beginning of the multibyte string that contains *c*.

```
char *MPrevChar(const char *s, const char *c) const
```

If there is an error, this method returns `NULL`. Call ITLocale::GetError() to retrieve a specific error message.

Related reference

Error return method on page 93

Index

W