Contents

Preface..................................................................................................................................................................... xi
Audience..................................................................................................................................................................... xi
Related documents.................................................................................................................................................. xi
Conventions........................................................................................................................................................... xii
Documentation Accessibility............................................................................................................................... xiii
Technical support.................................................................................................................................................... xiii

What’s New.............................................................................................................................................................. xv
New Features for Release 11.2.1............................................................................................................................ xv

1 TTClasses Development Environment

Setting up TTClasses on UNIX .......................................................................................................................... 1-1
  Set UNIX environment variables .................................................................................................................... 1-1
  Compile TTClasses on UNIX .......................................................................................................................... 1-2
  Compilation options on UNIX .......................................................................................................................... 1-2
  Install TTClasses after compilation (UNIX only) ........................................................................................... 1-2

Setting up TTClasses on Windows ................................................................................................................... 1-3
  Set Windows environment variables ............................................................................................................. 1-3
  Compile TTClasses on Windows .................................................................................................................... 1-3
  Compilation options on Windows .................................................................................................................... 1-3

TTClasses compiler macros .................................................................................................................................. 1-4
  Flags for C++ I/O streams: TTC_USE_STRINGSTREAM, USE_OLD_CPP_STREAMS ........ 1-4
    Neither flag: for C++ I/O stream code with ostream ................................................................. 1-4
    TTC_USE_STRINGSTREAM: for newest C++ I/O stream code with ostringstream ............ 1-4
    USE_OLD_CPP_STREAMS: for old C++ I/O stream code ........................................................... 1-5
    TTDEBUG: Generate additional debugging and error checking logic ............................................. 1-5
    TT_64BIT: Use TTClasses with 64-bit TimesTen ............................................................................. 1-5
  Platform-specific compiler macros ................................................................................................................... 1-5
    GCC ....................................................................................................................................................... 1-5
    HPUX .................................................................................................................................................... 1-5
  Considerations when using an ODBC driver manager ................................................................................ 1-5
  About the TimesTen TTClasses demos ................................................................................................. 1-6

2 Understanding and Using TTClasses

Overview of TTClasses .......................................................................................................................................... 2-1
Using TTCmd, TTConnection, and TTConnectionPool.............................................. 2-1
Considering TimesTen features for access control ............................................ 2-4
Managing TimesTen connections .................................................................. 2-5
   About DSNs .................................................................................................. 2-5
   Connecting and disconnecting ................................................................. 2-5
   Access control for connections ............................................................... 2-6
      Connection method signatures for access control .................................. 2-6
      CREATE SESSION privilege for access control ....................................... 2-7
      XLA privilege for XLA connections ......................................................... 2-7
Managing TimesTen data ............................................................................. 2-7
   Binding parameters ................................................................................... 2-7
      Binding IN parameters ........................................................................... 2-8
      Registering parameters .......................................................................... 2-9
      Binding OUT or IN OUT parameters ...................................................... 2-10
      Binding duplicate parameters .............................................................. 2-12
   Working with REF CURSORs ................................................................. 2-13
   Working with rowids .................................................................................. 2-15
   Setting a timeout or threshold for executing SQL statements .................... 2-16
Using TTClasses logging .......................................................................... 2-16
Using TTClasses XLA .................................................................................. 2-17
   Acknowledging XLA updates without using transaction boundaries .......... 2-17
   Acknowledging XLA updates at transaction boundaries .......................... 2-18
   Access control impact on XLA ............................................................... 2-19
3 Class Descriptions.................................................................................. 3-1

Commonly used TTClasses ........................................................................ 3-1
   TTGlobal .................................................................................................... 3-1
      Usage.................................................................................................. 3-1
      Public members ................................................................................ 3-2
      Public methods .................................................................................. 3-2
         sqlhenv() .......................................................................................... 3-3
         disableLogging() .............................................................................. 3-3
         setLogStream() ............................................................................... 3-3
         setLogLevel() .................................................................................. 3-3
   TTStatus ............................................................................................... 3-3
      Usage.................................................................................................. 3-3
      Subclasses ......................................................................................... 3-4
         TTErrr ................................................................................................ 3-4
         TTWarning ....................................................................................... 3-4
      Public members ................................................................................ 3-5
      Public methods .................................................................................. 3-5
         ostream() .......................................................................................... 3-5
         resetErrors() ................................................................................... 3-5
         isConnectionInvalid() ..................................................................... 3-5
         throwError() .................................................................................. 3-5
   TTConnection ....................................................................................... 3-6
      Usage.................................................................................................. 3-6
System catalog classes

TTCatalogTable

Public methods for batch operations

Public methods

Public members

Private methods

System catalog classes

TTCatalog

Public members

Public methods

Public members

Public methods

TTCatalogTable

Public members

Public methods

Public members

getCatalog()
<table>
<thead>
<tr>
<th>Method/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTypeName()</td>
<td>3-36</td>
</tr>
<tr>
<td>getScale()</td>
<td>3-36</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>3-36</td>
</tr>
<tr>
<td>getName()</td>
<td>3-36</td>
</tr>
<tr>
<td>getTableType()</td>
<td>3-36</td>
</tr>
<tr>
<td>getNumSpecialColumns()</td>
<td>3-38</td>
</tr>
<tr>
<td>getUserTable()</td>
<td>3-38</td>
</tr>
<tr>
<td>isSystemTable()</td>
<td>3-38</td>
</tr>
<tr>
<td>getDataType()</td>
<td>3-40</td>
</tr>
<tr>
<td>getColumnName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getRadix()</td>
<td>3-40</td>
</tr>
<tr>
<td>getCollation()</td>
<td>3-40</td>
</tr>
<tr>
<td>getNumColumns()</td>
<td>3-40</td>
</tr>
<tr>
<td>getTableName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getNullability()</td>
<td>3-40</td>
</tr>
<tr>
<td>getLength()</td>
<td>3-40</td>
</tr>
<tr>
<td>isUnique()</td>
<td>3-40</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>3-40</td>
</tr>
<tr>
<td>getScale()</td>
<td>3-40</td>
</tr>
<tr>
<td>getType()</td>
<td>3-40</td>
</tr>
<tr>
<td>getTableName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getIndexOwner()</td>
<td>3-40</td>
</tr>
<tr>
<td>getIndexName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getUserTable()</td>
<td>3-40</td>
</tr>
<tr>
<td>isSystemTable()</td>
<td>3-40</td>
</tr>
<tr>
<td>getDataType()</td>
<td>3-40</td>
</tr>
<tr>
<td>getColumnName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getRadix()</td>
<td>3-40</td>
</tr>
<tr>
<td>getCollation()</td>
<td>3-40</td>
</tr>
<tr>
<td>getNumColumns()</td>
<td>3-40</td>
</tr>
<tr>
<td>getTableName()</td>
<td>3-40</td>
</tr>
<tr>
<td>getNullability()</td>
<td>3-40</td>
</tr>
<tr>
<td>getLength()</td>
<td>3-40</td>
</tr>
<tr>
<td>isUnique()</td>
<td>3-40</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>3-40</td>
</tr>
<tr>
<td>getScale()</td>
<td>3-40</td>
</tr>
<tr>
<td>getType()</td>
<td>3-40</td>
</tr>
<tr>
<td>getTableName()</td>
<td>3-40</td>
</tr>
</tbody>
</table>

**XLA classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLA classes</td>
<td>3-42</td>
</tr>
<tr>
<td>TTXlaPersistConnection</td>
<td>3-42</td>
</tr>
<tr>
<td>Usage</td>
<td>3-42</td>
</tr>
<tr>
<td>Public members</td>
<td>3-43</td>
</tr>
<tr>
<td>Public methods</td>
<td>3-43</td>
</tr>
<tr>
<td>Public members</td>
<td>3-43</td>
</tr>
<tr>
<td>Public methods</td>
<td>3-43</td>
</tr>
</tbody>
</table>
TTXlaColumn ................................................................................................................. 3-54
Usage ......................................................................................................................... 3-54
Public members .......................................................................................................... 3-55
Public methods ........................................................................................................... 3-55
getColName() ............................................................................................................. 3-55
getPrecision() .............................................................................................................. 3-55
setScale() .................................................................................................................... 3-55
getType() ..................................................................................................................... 3-55
getSysColNum() .......................................................................................................... 3-56
getUserColNum() ....................................................................................................... 3-56
isNullable() ................................................................................................................. 3-56
isPKColumn() ............................................................................................................. 3-56
isTTTimestamp() ...................................................................................................... 3-56
isUpdated() ............................................................................................................... 3-56
Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including ODBC (Open Database Connectivity), OCI (Oracle Call Interface), Oracle Pro*C/C++ (precompiler for embedded SQL and PL/SQL instructions in C or C++ code), and PL/SQL (Oracle procedural language extension for SQL).

The TimesTen C++ Interface Classes (TTClasses) library was written to provide an easy-to-use, high-performance interface to TimesTen. This C++ class library provides wrappers around the most common ODBC functionality.

This preface covers the following topics:

- Audience
- Related documents
- Conventions
- Documentation Accessibility
- Technical support

Audience

This guide is for application developers who administer and access TimesTen through C++.

In addition to familiarity with the particular programming interface you use, you should be familiar with TimesTen, SQL (Structured Query Language), database operations, and ODBC.

Related documents

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network:

http://www.oracle.com/technology/documentation/timesten_doc.html

Oracle documentation is also available on the Oracle Technology network. This may be especially useful for Oracle features that TimesTen supports but does not attempt to fully document, such as OCI and Pro*C/C++:

http://www.oracle.com/technology/documentation/database.html
In particular, these Oracle documents may be of interest:

- Oracle Call Interface Programmer’s Guide
- Pro*C/C++ Programmer’s Guide
- Oracle Database Globalization Support Guide
- Oracle Database Net Services Administrator’s Guide
- Oracle Database SQL Language Reference

This manual occasionally refers to ODBC APIs. ODBC API reference documentation is available from Microsoft or a variety of third parties. For example:


Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows refers to Windows 2000, Windows XP and Windows Server 2003. The term UNIX refers to Solaris, Linux, HP-UX, Tru64 and AIX.

This document uses the following text conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic</td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td>monospace</td>
<td>Monospace type indicates code, commands, URLs, class names, function names, method names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.</td>
</tr>
<tr>
<td>italic monospace</td>
<td>Italic monospace type indicates a variable in a code example that you must replace. For example:</td>
</tr>
<tr>
<td>{}</td>
<td>Curly braces indicate that you must choose one of the items separated by a vertical bar (</td>
</tr>
<tr>
<td>[ ]</td>
<td>Square brackets indicate that an item in a command line is optional.</td>
</tr>
<tr>
<td>. . .</td>
<td>An ellipsis (..) after an argument indicates that you may use more than one argument on a single command line.</td>
</tr>
<tr>
<td>%</td>
<td>The percent sign indicates the UNIX shell prompt.</td>
</tr>
<tr>
<td>#</td>
<td>The number (or pound) sign indicates the UNIX root prompt.</td>
</tr>
</tbody>
</table>

TimesTen documentation uses these variables to identify path, file and user names:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>install_dir</td>
<td>The path that represents the directory where the current release of TimesTen is installed.</td>
</tr>
<tr>
<td>TTinstance</td>
<td>The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique alphanumeric instance name. This name appears in the install path.</td>
</tr>
</tbody>
</table>
Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Technical support

For information about obtaining technical support for TimesTen products, go to the following Web address:

http://www.oracle.com/support/contact.html

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bits or bb</td>
<td>Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.</td>
</tr>
<tr>
<td>release or rr</td>
<td>Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.</td>
</tr>
<tr>
<td>DSN</td>
<td>The data source name.</td>
</tr>
</tbody>
</table>
What's New

This section summarizes new features and functionality of Oracle TimesTen In-Memory Database Release 11.2.1 that are documented in this guide, providing links into the guide for more information.

New Features for Release 11.2.1

TimesTen Release 11.2.1 includes the following new features covered in this guide.

Features for Release 11.2.1.6.0

TTCclasses implements the following features beginning with the TimesTen Release 11.2.1.6.0.

- OUT parameters
  
  Discussion of binding parameters includes new support for binding OUT and IN OUT parameters.
  
  See appropriate subsections under “Binding parameters” on page 2-7.

- Duplicate parameters
  
  TimesTen supports either of two modes for binding duplicate parameters in a SQL statement. Use the `DuplicateBindMode` general connection attribute to choose between Oracle mode (now the default) and traditional TimesTen mode.
  
  See “Binding duplicate parameters” on page 2-12.

- REF CURSORs
  
  `REF CURSOR` is a PL/SQL concept, where a `REF CURSOR` is a handle to a cursor over a SQL result set and can be passed between PL/SQL and an application.
  
  See “Working with REF CURSORs” on page 2-13.

- Rowids
  
  Each row in a TimesTen database table has a unique identifier known as its rowid. TimesTen now supports Oracle-style rowids. An application can retrieve the rowid of a row from the `ROWID` pseudocolumn. Rowids can be represented in either binary or character format.
  
  See “Working with rowids” on page 2-15.

- DML returning (RETURNING INTO clause)
  
  TimesTen now supports the RETURNING INTO clause, referred to as DML returning, with an INSERT, UPDATE, or DELETE statement to return specified...
items from a row that was affected by the action. This is included in the discussion of OUT parameters in "Binding OUT or IN OUT parameters" on page 2-10.

- Exception handling

  By default, TTStatus objects are thrown as exceptions whenever an error occurs. This allows C++ applications to use \texttt{try/catch} blocks to detect and recover from failure, which is the recommended mode of operation. The TTEXCEPT flag, which allowed exceptions to be disabled in previous releases, is now deprecated. It is possible, however, to selectively suppress exceptions and manually check a TTStatus object for error conditions by initializing the TTStatus object with the value \texttt{TTStatus::DO\_NOT\_THROW}, then passing that object as the last parameter of a method call. Most TTClasses methods documented in this manual also support a signature with this TTStatus parameter as the last parameter in the calling sequence, although these signatures are not documented and it is generally not recommended to operate in this way. See "TTStatus" on page 3-3.

- API changes

  Be aware that there have been numerous method additions and changes, especially regarding TTStatus parameters in the calling sequences. Consult the documentation in Chapter 3, "Class Descriptions," carefully. Many methods were documented with a TTStatus parameter in previous releases, and while these are still supported for backward compatibility, using these methods is no longer documented or encouraged.

**Features for Release 11.2.1.0**

- Quick Start demos

  The 11.2.1 release includes an optional Quick Start feature with introductory information, tutorials, and new or reworked demo applications. Note that the demos are in a different location than in earlier releases and some have been renamed. See "About the TimesTen TTClasses demos" on page 1-6 and \texttt{install_dir/quickstart.html} in your installation.

- Access control

  Perhaps the most significant overall change to previous functionality in TimesTen Release 11.2.1 is access control. TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, and sequences. This also affects access to certain TimesTen built-in procedures, utilities, and connection attributes. See "Considering TimesTen features for access control" on page 2-4. For general information, see "Managing Access Control" in Oracle TimesTen In-Memory Database Operations Guide.
This chapter provides information to help you get started with your TTClasses development environment.

TTClasses comes compiled and preconfigured during TimesTen installation. If you have a different C++ runtime than what TTClasses was compiled with, recompile the library using the `make` (UNIX) or `nmake` (Microsoft Windows) utility.

The information here includes a discussion of how to set environment variables, a description of the compilation process, and an introduction to the Quick Start demo applications for TTClasses. The following topics are covered:

- Setting up TTClasses on UNIX
- Setting up TTClasses on Windows
- TTClasses compiler macros
- Considerations when using an ODBC driver manager
- About the TimesTen TTClasses demos

### Setting up TTClasses on UNIX

This section covers the following topics for setting up TTClasses in a UNIX environment:

- Set UNIX environment variables
- Compile TTClasses on UNIX
- Compilation options on UNIX
- Install TTClasses after compilation (UNIX only)

### Set UNIX environment variables

To use TTClasses, ensure that your shell environment variables are set correctly. You can optionally source one of the following scripts or add a line to source one of these scripts in your login initialization script (`profile` or `.cshrc`), where `install_dir` is the installation directory for your TimesTen instance:

```bash
install_dir/bin/ttenv.sh  # sh/ksh/bash
install_dir/bin/ttenv.csh  # csh/tcsh
```
Compile TTClasses on UNIX

If you have application linking problems, which can be caused by using a different C++ runtime than what TTClasses was compiled with, recompile the library using the make utility.

To recompile TTClasses, change to the ttclasses directory, where install_dir is the installation directory for your TimesTen instance:

```
$ cd install_dir/ttclasses
```

Run `make clean` for a fresh start:

```
$ make clean
```

You can recompile TTClasses for both direct mode and client/server as follows:

```
$ make
```

Alternatively, to compile TTClasses for client/server only, use the MakefileCS Makefile:

```
$ make -f MakefileCS
```

Compilation options on UNIX

The following 'make target' options are available when you compile TTClasses in a UNIX environment:

- **all**: Build a shared optimized library or libraries (default). When used with Makefile this can be for either direct mode or client/server. When used with MakefileCS this is for client/server only.
- **shared_opt**: Build a shared optimized library. Currently this has the same effect as all.
- **shared_debug**: Build a shared debug library.
- **static_opt**: Build a static optimized library.
- **static_debug**: Build a static debug library.
- **opt**: Build the optimized libraries (shared and static).
- **debug**: Build the debug libraries (shared and static).
- **clean**: Delete the TTClasses libraries and object files.

To specify a make target, use the name of the make target on the command line. For example, to build a shared debug version of TTClasses:

```
$ make clean
$ make shared_debug
```

Install TTClasses after compilation (UNIX only)

After compilation, install the library so all users of the TimesTen instance can use TTClasses. The following shows the steps to install the TTClasses library on a UNIX system:

```
$ cd install_dir/ttclasses
$ make install
```
Setting up TTClasses on Windows

This section covers the following topics for setting up TTClasses in a Windows environment:

- Set Windows environment variables
- Compile TTClasses on Windows
- Compilation options on Windows

**Note:** Installing TTClasses after compiling the TTClasses library happens automatically on Windows.

Set Windows environment variables

Before recompiling, ensure that the PATH, INCLUDE, and LIB environment variables point to the correct Visual Studio directories. Execute the applicable Visual Studio C++ batch file (for example, VCVARS32.BAT or VSVARS32.BAT) to accomplish this.

Then set environment variables for TimesTen (if they were not already set during installation) by running the following:

```
install_dir\bin\ttenv.bat
```

Compile TTClasses on Windows

If you have application linking problems, which can be caused by using a different C++ runtime than what TTClasses was compiled with, recompile the library using the `nmake` utility.

To recompile TTClasses, change to the `ttclasses` directory, where `install_dir` is the installation directory for your TimesTen instance:

```
install_dir\ttclasses
```

Run `nmake clean` for a fresh start:

```
install_dir\ttclasses> nmake clean
```

Then recompile. By default this is for both direct mode and client/server:

```
install_dir\ttclasses> nmake
```

Compilation options on Windows

The following `make target` options are available when you compile TTClasses in a Windows environment:

- **all:** Build shared optimized libraries for direct mode and client/server (default).
- **client:** Build shared optimized library for client/server only.
- **msdm:** Build shared optimized library for Microsoft driver manager.
- **clean:** Delete the TTClasses libraries and object files.

To specify a make target, use the name of the make target on the command line.

For example, to build only the client/server TTClasses library:

```
install_dir\ttclasses> make clean
install_dir\ttclasses> make client
```
TTClasses compiler macros

Most users do not need to manipulate the TTClasses Makefile. If you need to modify the TTClasses Makefile manually, you can add flags for the TTClasses compiler macros to the Makefile. For UNIX, add `-Dflagname`. For Windows, add `/Dflagname`.

This section includes information about the following compiler macros:

- **Flags for C++ I/O streams**: TTC_USE_STRINGSTREAM, USE_OLD_CPP_STREAMS
- **TTDEBUG**: Generate additional debugging and error checking logic
- **TT_64BIT**: Use TTClasses with 64-bit TimesTen

Also see 'Platform-specific compiler macros' on page 1-5.

### Flags for C++ I/O streams: TTC_USE_STRINGSTREAM, USE_OLD_CPP_STREAMS

There are multiple types of C++ streams and they are not compatible with each other. TimesTen provides two related flags. The types of streams you use in your application determine which flag to use, or whether you should use neither, as follows:

- For relatively new (but not the newest) types of streams, where you are including `<iostream>` and using the `ostrstream` class, use neither flag. This is the default for most platforms and compilers.
- For the newest types of streams, where you are including `<iostream>` and using the `ostringstream` class, use the TTC_USE_STRINGSTREAM flag.
- For old types of streams, where you are including `<iostream.h>` and using the `ostrstream` class, use the USE_OLD_CPP_STREAMS flag. This is the default for some older platforms and compilers.

Check your TTClasses Makefile to see if either flag is set. If the flags are not set properly, then update the Makefile as appropriate, recompile TTClasses, and replace the previous TTClasses library file with the recompiled one.

Further information is provided in the subsections that follow.

#### Neither flag: for C++ I/O stream code with ostrstream

As noted above, if you are using C++ I/O stream code where you are including `<iostream>` and using the `ostrstream` class, neither the TTC_USE_STRINGSTREAM flag nor the USE_OLD_CPP_STREAMS flag should be set. If either is set in the TTClasses Makefile, remove the line and recompile TTClasses.

#### TTC_USE_STRINGSTREAM: for newest C++ I/O stream code with ostringstream

This compiler flag is for use with C++ compilers that reliably support the newest C++ stream types and the `ostringstream` class. If your program uses the newest stream types, then TTClasses must be compiled with the `-DTTC_USE_STRINGSTREAM` setting. This allows TimesTen to use `ostringstream` and `<iostream>` for compatibility with your program code.

Also note that if your program uses the newest C++ streams, the USE_OLD_CPP_STREAMS flag must not be set. If the TTClasses Makefile has the wrong settings, correct them and recompile TTClasses.
USE_OLD_CPP_STREAMS: for old C++ I/O stream code
This compiler flag is for older C++ compilers that do not support `<iostream>`. If your program uses old C++ streams, where you include `<iostream.h>` and use `ostrstream`, then TTClasses must be compiled with the `-DUSE_OLD_CPP_STREAMS` setting to be compatible with your program code.

Also note that if your program uses old C++ streams, the TTC_USE_STRINGSTREAM flag must not be set. If the TTClasses Makefile has the wrong settings, correct them and recompile TTClasses.

TTDEBUG: Generate additional debugging and error checking logic
Compile TTClasses with `-DTTDEBUG` to generate extra debugging information. This extra information reduces performance somewhat, so use this flag only in development (not production) systems.

TT_64BIT: Use TTClasses with 64-bit TimesTen
Compile TTClasses with `-DTT_64BIT` if you are writing a 64-bit TimesTen application.

Note that 64-bit TTClasses has been tested on AIX, HP-UX, Solaris, Red Hat Linux, and Tru64.

Platform-specific compiler macros
The following compiler macros are specific to a particular platform or compiler combination. You should not have to specify these compiler macros manually. Their use is determined by the Makefile chosen by the configure program.

GCC
Compile TTClasses with the `-DGCC` flag when using gcc on any platform.

HPUX
Compile TTClasses with the `-DHPUX` flag when compiling on HP-UX.

Considerations when using an ODBC driver manager
Be aware of the following limitations in TTClasses when you use an ODBC driver manager. (These restrictions do not apply to the demo ttdm driver manager supplied with the TimesTen Quick Start.)

- XLA functionality does not work.
- REF CURSOR functionality does not work.
- The following methods of the TTCmd class do not work.

```c
TTCmd::getColumn(int cno, SQL_BIGINT* iP)
TTCmd::getColumn(int cno, SQL_TINYINT* iP)
TTCmd::getColumn(int cno, SQL_WCHAR** wcPP)
```
About the TimesTen TTClasses demos

After you have configured your C++ environment, you can confirm that everything is set up correctly by compiling and running the TimesTen Quick Start demo applications. Refer to the Quick Start welcome page at install_dir/quickstart.html, especially the links under Sample Programs, for information about the following:

- Demo schema and setup: The build_sampledb script creates a sample database and demo schema. You must run this before you start using the demos.
- Demo environment and setup: The ttquickstartenv script, a superset of the ttenv script generally used for TimesTen setup, sets up the demo environment. You must run this each time you enter a session where you want to compile and run any of the demos.
- Demos and setup: TimesTen provides demos for TTClasses and XLA in subdirectories under the install_dir/quickstart/sample_code directory. For instructions on compiling and running the demos, see the README files in the subdirectories.
- What the demos do: A synopsis of each demo is provided.

```cpp
TTCmd::getColumn(int cno, SQLCHAR** wcPP, int* byteLenP)
TTCmd::getColumnNullable(int cno, SQLBIGINT* iP)
TTCmd::getColumnNullable(int cno, SQLTINYINT* iP)
TTCmd::getColumnNullable(int cno, SQLCHAR** wcPP)
TTCmd::getColumnNullable(int cno, SQLCHAR** wcPP, int* byteLenP)
TTCmd::getNextColumn(SQLBIGINT* iP)
TTCmd::getNextColumn(SQLTINYINT* iP)
TTCmd::getNextColumn(SQLCHAR** wcPP)
TTCmd::getNextColumn(SQLCHAR** wcPP, int* byteLenP)
TTCmd::getNextColumnNullable(SQLBIGINT* iP)
TTCmd::getNextColumnNullable(SQLTINYINT* iP)
TTCmd::getNextColumnNullable(SQLCHAR** wcPP)
TTCmd::getNextColumnNullable(SQLCHAR** wcPP, int* byteLenP)
TTCmd::setParam(int pno, SQLBIGINT value)
TTCmd::setParam(int pno, SQLTINYINT value)
TTCmd::setParam(int pno, SQLCHAR* valueP, int byteLen);
TTCmd::getParam(int pno, SQLBIGINT* iP)
TTCmd::getParam(int pno, SQLTINYINT* iP)
TTCmd::getParam(int pno, SQLCHAR** wcPP)
TTCmd::getParam(int pno, SQLCHAR** wcPP, int* byteLenP)
TTCmd::BindParameter(int pno, u_short batSz, SQLBIGINT* user_biP)
TTCmd::BindParameter(int pno, u_short batSz, SQLTINYINT* user_tiP)
TTCmd::BindParameter(int pno, u_short batSz, SQLCHAR** user_wcP,
                     SQLLEN* userByteLenP, size_t maxByteLen)
```
Understanding and Using TTClasses

This chapter provides some general overview and best practices for TTClasses as well as descriptions of key features added in the TimesTen 11.2.1 release. It includes the following topics:

- Overview of TTClasses
- Using TTCmd, TTConnection, and TTConnectionPool
- Considering TimesTen features for access control
- Managing TimesTen connections
- Managing TimesTen data
- Using TTClasses logging
- Using TTClasses XLA

Overview of TTClasses

The TimesTen C++ Interface Classes library (TTClasses) provides wrappers around the most common ODBC functionality to allow access to TimesTen data stores. It was developed to meet the demand for an API that is easier to use than ODBC but does not sacrifice performance. Refer to ODBC API reference documentation for detailed information about ODBC.

In addition to providing a C++ interface to the TimesTen ODBC interface, TTClasses supplies an interface to the TimesTen’s Transaction Log API (XLA). XLA allows an application to monitor one or more tables in a TimesTen data store. When other applications change that table, the changes are reported through XLA to the monitoring application. TTClasses provides an easy-to-use interface to the most commonly used aspects of XLA functionality. For general information about XLA, see “XLA and TimesTen Event Management” in Oracle TimesTen In-Memory Database C Developer’s Guide.

TTClasses is also intended to promote best practices when writing application software that uses the TimesTen Data Manager. The library uses TimesTen in an optimal manner. For example, autocommit is disabled by default. Parameterized SQL is strongly encouraged and its use is greatly simplified in TTClasses compared to hand-coded ODBC.

Using TTCmd, TTConnection, and TTConnectionPool

While TTClasses can be used in a number of ways, the following general approach has been used successfully and can easily be adapted to a variety of applications.
To achieve optimal performance, real-time applications should use prepared SQL statements. Ideally, all SQL statements that will be used by an application are prepared when the application begins, using a separate TTCmd object for each statement. In ODBC (and thus in the TTClasses), statements are bound to a particular connection, so a full set of the statements used by the application will often be associated with every connection to the TimesTen database.

An easy way to accomplish this is to develop an application-specific class that is derived from TTConnection. For an application called XYZ, you can create a class called XYZConnection, for example. The XYZConnection class contains private TTCmd members representing the prepared SQL statements that can be used in the application, and provides new public methods to implement the application-specific database functionality through these private TTCmd members.

Before a TTCmd object can be used, a SQL statement (such as SELECT, INSERT, UPDATE, or DELETE) must be associated with it. The association is accomplished by using the Prepare() method, which also compiles and optimizes the SQL statement to ensure it will be executed in an efficient manner. Note that the Prepare() method only prepares and does not execute the statement.

With TimesTen, statements are typically parameterized for better performance. Consider the following SQL statements:

```
SELECT col1 FROM table1 WHERE C = 10;
SELECT col1 FROM table1 WHERE C = 11;
```

It is more efficient to prepare a single parameterized statement and execute it multiple times:

```
SELECT col1 FROM table1 WHERE C = ?;
```

The value for "?" is specified at runtime by using the TTCmd::setParam() method. There is no need to explicitly bind columns or parameters to a SQL statement, as is necessary when you use ODBC directly. TTCmd automatically defines and binds all necessary columns at prepare time. Parameters are bound at execution time. Note that preparing is a relatively expensive operation. When an application establishes a connection to TimesTen, using TTConnection::Connect(), the application should prepare all TTCmd objects associated with the connection.

A TTStatus object is thrown as an exception if an error occurs during the prepare operation. In general, anytime a TTClasses method encounters an error, it throws an exception in this way, which the application should catch and handle appropriately. The TTClasses Quick Start demo applications show examples of how this is done. (See "About the TimesTen TTClasses demos" on page 1-6.)

**Example 2–1 Definition of a connection class**

This is an example of a class that inherits from TTConnection.

```cpp
class XYZConnection : public TTConnection {
    private:
```

Note: If TTConnection and TTCmd lack any getter or setter methods you need, you can access underlying ODBC connection and statement handles directly, through the TTConnection::getHdbc() and TTCmd::getHandle() methods. Similarly, there is a TTGlobal::sqlhenv() method to access the ODBC environment handle.
Using TTCmd, TTConnection, and TTConnectionPool

In this example, an XYZConnection object is a connection to TimesTen that can be used to perform three application-specific operations: addUser(), updateUser(), and queryUser(). These operations are specific to the XYZ application. The implementation of these three methods can use the TTCmd objects to implement the database operations of the application.

Example 2–2 Definition of a Connect() method

This example shows an implementation of the XYZConnection::Connect() method.

```cpp
void XYZConnection::Connect(const char* connStr, const char* user, const char* pwd) {
    try {
        TTConnection::Connect(connStr, user, pwd);
        updateData.Prepare(this, "update mydata v set foo = ? where bar = ?");
        insertData.Prepare(this, "insert into mydata values(?,0)"),
        queryData.Prepare(this, "select i from mydata where name = ?");
    } catch (TTStatus st) {
        cerr << "Error in XYZConnection::Connect: " << st << endl;
        return;
    }
}
```

This Connect() method makes the XYZConnection object and its application-specific methods fully operational.

This approach also works well with the design of the TTConnectionPool class. The application can create numerous objects of type XYZConnection and add them to a TTConnectionPool object. By calling TTConnectionPool::ConnectAll(), the application connects all connections in the pool to the database as well as preparing all SQL statements. Refer to the usage discussion under "TTConnectionPool" on page 3-11, which includes important information.

This application design allows database access to be easily separated from the application business logic. Only the XYZConnection class contains database-specific code.
Examples of this application design can be found in several of the TTClasses sample programs that are included with the TimesTen Quick Start. See “About the TimesTen TTClasses demos” on page 1-6.

Note that other configurations are possible. Some customers have extended this scheme further, so that SQL statements to be used in an application are listed in a table in the database, rather than being hard-coded in the application itself. This allows changes to database functionality to be implemented by making database changes rather than application changes.

**Example 2-3  Definition of a Disconnect() method**

This example shows an implementation of the `XYZConnection::Disconnect()` method.

```cpp
void XYZConnection::Disconnect()
{
  updateData.Drop();
  insertData.Drop();
  queryData.Drop();

  TTConnection::Disconnect();
}
```

---

**Considering TimesTen features for access control**

TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. You can refer to “Managing Access Control” in Oracle TimesTen In-Memory Database Operations Guide for general information about these features. Also see “Considering TimesTen features for access control” in Oracle TimesTen In-Memory Database C Developer’s Guide.

For any query, SQL DML statement, or SQL DDL statement discussed in this document or used in an example, it is assumed that the user has appropriate privileges to execute the statement. For example, a SELECT statement on a table requires ownership of the table, SELECT privilege granted on the table, or the SELECT ANY TABLE system privilege.

Refer to “SQL Statements” in Oracle TimesTen In-Memory Database SQL Reference for the privilege required for any given SQL statement.

Privileges are granted through the SQL statement GRANT and revoked through the SQL statement REVOKE. Some privileges are granted to all users through the PUBLIC role, of which each user is a member. See “The PUBLIC role” in Oracle TimesTen In-Memory Database SQL Reference for information about that role.

In addition, access control affects connecting to a data store (as discussed in “Access control for connections” on page 2-6), setting connection attributes, using XLA (as discussed in “Access control impact on XLA” on page 2-20), and executing C utility functions.
Managing TimesTen connections

This section covers topics related to connecting to a TimesTen database:

- About DSNs
- Connecting and disconnecting
- Access control for connections

About DSNs

Oracle TimesTen In-Memory Database Operations Guide contains information about creating a DSN (data source name) for a TimesTen data store. The type of DSN you create depends on whether your application will connect directly to the data store or will connect by a client.

If you intend to connect directly to the data store, refer to “Managing TimesTen Data Stores” in Oracle TimesTen In-Memory Database Operations Guide. There are sections on creating a DSN for a direct connection from UNIX or Windows.

If you intend to create a client connection to the data store, refer to “Working with the TimesTen Client and Server” in Oracle TimesTen In-Memory Database Operations Guide. There are sections on creating a DSN for a client/server connection from UNIX or Windows.

Note: A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating a child process, the child must not use the connection.

Connecting and disconnecting

Based on the XYZConnection class discussed in “Using TTCmd, TTConnection, and TTConnectionPool” on page 2-1, you could connect to and disconnect from TimesTen as shown in the following example.

Example 2-4 Connecting to and disconnecting from TimesTen

```java
...  
XYZConnection conn;  
char connStr[256];  
char use[30];  
char pwd[10];  
...  
try {  
    conn.Connect(connStr, use, pwd);  
}  
```
Managing TimesTen connections

catch (TTWarning st) {
    cerr << "Warning connecting to TimesTen: " << st << endl;
}
catch (TTError st) {
    cerr << "Error connecting to TimesTen: " << st << endl;
    exit(1);
}

// ... Work with the database connection...

try {
    conn.Disconnect();
}
catch (TTStatus st) {
    cerr << "Error disconnecting from TimesTen: " << st << endl;
    exit(1);
}

Access control for connections

This section covers access control features related to how you connect to the database with TTClasses.

For a general access control overview, refer to "Considering TimesTen features for access control" on page 2-4.

Connection method signatures for access control

The following method signatures are defined for the TTConnection, TTConnectionPool, and TTXlaPersistConnection classes. (Note that in all cases, signatures are also supported with a TTStatus object as the last parameter, but using the methods with TTStatus is not typical.)

virtual void TTConnection::Connect(const char* connStr)

virtual void TTConnection::Connect(const char* connStr, const char* username, const char* password)

virtual void TTConnection::Connect(const char* connStr, DRIVER_COMPLETION_ENUM driverCompletion)

void TTConnectionPool::ConnectAll(const char* connStr)

void TTConnectionPool::ConnectAll(const char* connStr, const char* username, const char* password)

virtual void TTXlaPersistConnection::Connect(const char* connStr, const char* username, const char* password, const char* bookmarkStr, bool createBookmarkFlag)

virtual void TTXlaPersistConnection::Connect(const char* connStr, DRIVER_COMPLETION_ENUM driverCompletion, const char* bookmarkStr, bool createBookmarkFlag)
virtual void TTXlaPersistConnection::Connect(const char* connStr, const char* username, const char* password, const char* bookmarkStr)

virtual void TTXlaPersistConnection::Connect(const char* connStr, DRIVER_COMPLETION_ENUM driverCompletion, const char* bookmarkStr)

Notes:
- The connection string (connStr value) can specify the user name and password, such as 'DSN=testdb;uid=brian;pwd=welcome'. But note that for signatures that take connection string, user name, and password arguments, the user name and password arguments take precedence over any user name or password specified in the connection string.
- See "TTConnection" on page 3-6 for information about DRIVER_COMPLETION_ENUM values.

CREATE SESSION privilege for access control
Privilege to connect to a TimesTen data store must be explicitly granted to every user other than the instance administrator, through the CREATE SESSION privilege. This is a system privilege. It must be granted by an administrator to the user, either directly or through the PUBLIC role. Refer to "Managing Access Control" in Oracle TimesTen In-Memory Database Operations Guide for additional information and examples.

XLA privilege for XLA connections
In addition to the CREATE SESSION privilege, a user must be granted the XLA privilege to create an XLA connection and execute XLA functionality, as noted in "Access control impact on XLA" on page 2-20.

Managing TimesTen data
This section covers the following topics for working with data in a TimesTen database.
- Binding parameters
- Working with REF CURSORs
- Working with rowids
- Setting a timeout or threshold for executing SQL statements

Binding parameters
This section discusses parameter binding for SQL statements. The TTCmd class supplies the methods setParam() and BindParameter() (for batch operations) to bind parameters. It also supplies the method registerParam() to support output and input/output parameters or to override default bind types. There is also functionality to support either possible TimesTen DuplicateBindMode setting if there are duplicate parameters.

These topics are covered in the following sections.
Binding IN parameters

For non-batch operations, use the `TTCmd::setParam()` method to bind IN parameters for SQL statements, specifying the parameter position and the value to be bound. For batch operations, use the `TTCmd::BindParameter()` method. (See Example 3-5, "Using the ExecuteBatch() method" on page 3-31 for an example of batch operations.)

For non-batch operations, Example 2-5 shows snippets from a class `SampleConnection`, where parameters are bound to insert a row into a table. (This example is from the TimesTen Quick Start demo `basics.cpp`. See "About the TimesTen TTClasses demos" on page 1-6.) Implementation of the `Connect()` method is omitted here, but see Example 2-2 on page 2-3 for a `Connect()` implementation.

Assume a table `basics` defined as follows:

```sql
create table basics (name char(10) not null primary key, i tt_integer);
```

Example 2-5  Binding parameters to insert a row (non-batch)

```cpp
class SampleConnection : public TTConnection
{
using TTConnection::Connect;

private:
TTCmd        insertData;
...

protected:
public:
SampleConnection();
~SampleConnection();
virtual void Connect(const char* connStr,
DRIVER_COMPLETION_ENUM driverCompletion);
void insert(char* nameP);
...
...
// Assume a Connect() method implemented with the following:
// insertData.Prepare(this, "insert into basics values(:name, :value);";
... 
}

void SampleConnection::insert(char* nameP)
{
static long i = 0;
insertData.setParam(1, nameP);
insertData.setParam(2, i++);
insertData.Execute();
}
```
Managing TimesTen data

```c
int main(int argc, char** argv)
{

    char name[10];
    SampleConnection conn;

    // Assume conn set as a connection, name as a character string.
    try {
        conn.insert(name);
    }
    catch (TTStatus st) {
        cerr << "Error inserting row " << name << ":" << st << endl;
        conn.Rollback();
    }
}
```

Registering parameters

The `TTCmd` class provides the method `registerParam()`, which allows you to specify the SQL type, precision, and scale of a parameter (as applicable) and whether the parameter is IN, OUT, or IN OUT. A `registerParam()` call is required for an OUT or IN OUT parameter, which could be a REF CURSOR (OUT only) or a parameter from a PL/SQL RETURNING INTO clause (OUT only), procedure, or function.

For an IN parameter, `TTClasses` by default derives the SQL type from the bound C type for the `setParam()` or `bindParameter()` call according to the mappings shown in Table 2–1. It is not typical to need a `registerParam()` call for an IN parameter, but you can call it if you have reason to use a particular SQL type or precision or scale.

**Table 2–1  TTClasses C type to SQL type mappings**

<table>
<thead>
<tr>
<th>C type</th>
<th>SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>const char*</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>const void*</td>
<td>SQL_VARBINARY</td>
</tr>
<tr>
<td>double</td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>DATE_STRUCT</td>
<td>SQL_DATE</td>
</tr>
<tr>
<td>float</td>
<td>SQL_REAL</td>
</tr>
<tr>
<td>int</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>SQL_BIGINT</td>
<td>SQL_BIGINT</td>
</tr>
<tr>
<td>SQL_CHAR*</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>SQL_INTEGER</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>SQL_SMALLINT</td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td>SQL_TINYINT</td>
<td>SQL_TINYINT</td>
</tr>
</tbody>
</table>
A `registerParam()` call can be either before or after the related `setParam()` or `bindParam()` call and takes precedence regarding SQL type, precision, and scale (as applicable).

The method signature is as follows:

```c
inline void TTCmd::registerParam(int pno,
int inputOutputType,
int sqltype,
int precision = 0,
int scale = 0)
```

- `pno` is the parameter position in the statement.
- `inputOutputType` can be `TTCmd::PARAM_IN`, `TTCmd::PARAM_OUT`, or `TTCmd::PARAM_INOUT`.
- `sqltype` is the SQL type of the data (for example, `SQLINTEGER`).
- `precision` and `scale` (both optional) are used the same way as in an ODBC `SQLBindParameter` call. For primitive types, `precision` and `scale` settings are ignored.

**Note:** See the next section, "Binding OUT or IN OUT parameters", for an example. Also see "registerParam()" on page 3-19 for additional reference information.

### Table 2–1 (Cont.) TTClasses C type to SQL type mappings

<table>
<thead>
<tr>
<th>C type</th>
<th>SQL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLWCHAR*</td>
<td>SQL_NVARCHAR</td>
</tr>
<tr>
<td>TIME_STRUCT</td>
<td>SQL_TIME</td>
</tr>
<tr>
<td>TIMESTAMP_STRUCT</td>
<td>SQL_TIMESTAMP</td>
</tr>
</tbody>
</table>

**Binding OUT or IN OUT parameters**

Beginning in Release 11.2.1.6.0, TTClasses supports output and input/output parameters. This includes REF CURSORS (OUT only), parameters from a PL/SQL procedure or function that has OUT or IN OUT parameters, or a parameter from a RETURNING INTO clause (OUT only).

You must use the `TTCmd::registerParam()` method, described in the preceding section, to inform TTClasses if a parameter in a SQL statement is OUT or IN OUT. For the `bindParam` setting in the method call, use `TTCmd::PARAM_OUT` or `TTCmd::PARAM_INOUT` as appropriate.

For non-batch operations, after the SQL statement has been executed, use the appropriate `TTCmd::bindParam()` method to retrieve the output value, specifying the parameter position and the variable into which the value is placed. There is a signature for each data type.

For batch operations, `TTCmd::bindParam()` is used for OUT or IN OUT parameters as well as for IN parameters, in either case before the statement is executed. After statement execution, the data for an OUT value will be in the buffer specified in the `bindParam` call. `bindParam()` has a signature for each data type. Note that for an IN OUT parameter in batch operations, `bindParam()` is called only once, before statement execution. Before execution
the specified buffer contains the input, and after statement execution it contains the output.

The following examples provide code fragments showing the use of OUT and IN OUT parameters.

**Example 2–6 Using IN and OUT parameters (non-batch)**

This example uses input and output parameters. The `setParam()` call binds the value of the input parameter :a. The `getParam()` call retrieves the value of the output parameter :b. The output parameter is also registered as required.

```c++
// t1 has a single TT_INTEGER column
cmd.Prepare(&conn, "insert into t1 values (:a) returning c1 into :b");
cmd.setParam(1, 99);
cmd.registerParam(2, TTCmd::PARAM_OUT, SQLINTEGER);
cmd.Execute();
SQLINTEGER outval;
if (cmd.getParam(2, &outval))
  cerr << "The output value is null." << endl;
else
  cerr << "The output value is " << outval << endl;
...
```

**Example 2–7 Using IN and OUT parameters (batch operations)**

This example uses input and output parameters in a batch operation. The first `BindParameter()` call provides the input data for the first parameter :a. The second `BindParameter()` call provides a buffer for output data for the second parameter :b.

```c++
#define BATCH_SIZE 5
int input_int_array[BATCH_SIZE] = { 91, 92, 93, 94, 95 };
int output_int_array[BATCH_SIZE] = { -1, -1, -1, -1, -1 };
int numrows;

// t1 has a single TT_INTEGER column
cmd.PrepareBatch(&conn, "insert into t1 values (:a) returning c1 into :b", Batch_SIZE);
cmd.BindParameter(1, BATCH_SIZE, input_int_array);
cmd.BindParameter(2, BATCH_SIZE, output_int_array);
cmd.registerParam(2, TTCmd::PARAM_OUT, SQL_INTEGER);
numrows = cmd.ExecuteBatch(BATCH_SIZE);
...
```

**Example 2–8 Using IN OUT parameters**

This example uses an IN OUT parameter. It is registered as required. The `setParam()` call binds its input value and the `getParam()` call retrieves its output value.

```c++
// this example uses an IN OUT parameter:
// it is registered as required.
// the setParam() call binds its input value
// and the getParam() call retrieves its output value.
... 
```

Understanding and Using TTCclasses 2-11
cerr << "The output value is null." << endl;
else
  cerr << "The output value is " << outval << endl;
...

Example 2-9 Using OUT and IN OUT parameters

This example uses OUT and IN OUT parameters. Assume a PL/SQL procedure as follows:

```plsql
create or replace procedure my_proc ( 
a in number,
b in number,
c out number,
d in out number) as
begin
  c := a + b;
d := a + b - d;
end my_proc;
```

The input parameters for the procedure are taken as constants in this example rather than as bound parameters, so only the OUT parameter and IN OUT parameter are bound. Both are registered as required. The `setParam()` call provides the input value for the IN OUT parameter :var1. The first `getParam()` call retrieves the value for the output parameter :sum. The second `getParam()` call retrieves the output value for the IN OUT parameter :var1.

```plsql
... 
cmd.Prepare(&conn, "begin my_proc (10, 5, :sum, :var1); end;"");
cmd.registerParam (1, TTCmd::PARAM_OUT, SQL_DECIMAL, 38);
cmd.registerParam (2, TTCmd::PARAM_INOUT, SQL_DECIMAL, 38);
cmd.setParam(2, "99");
cmd.Execute();

SQLINTEGER outval1, outval2;
if (cmd.getParam(1, &outval1))
  cerr << "The first output value is null." << endl;
else
  cerr << "The first output value is " << outval << endl;
if (cmd.getParam(2, &outval2))
  cerr << "The second output value is null." << endl;
else
  cerr << "The second output value is " << outval << endl;
...

Binding duplicate parameters

TimesTen supports two modes for binding duplicate parameters in a SQL statement. In Oracle mode, where `DuplicateBindMode=0` (the default), multiple instances of the same parameter name are considered to be distinct parameters. In traditional TimesTen mode, where `DuplicateBindMode=1`, multiple instances of the same parameter name are considered to be the same parameter (as in earlier TimesTen releases).
For illustration, considering the following query:

```
SELECT * FROM employees
WHERE employee_id < :a AND manager_id > :a AND salary < :b;
```

In Oracle mode, when parameter position numbers are assigned, a number is given to each parameter occurrence without regard to name duplication. The application must, at a minimum, bind a value for the first occurrence of each parameter name. For any subsequent occurrence of a given parameter name, the application can bind a different value for the occurrence or it can leave the parameter occurrence unbound. In the latter case, the subsequent occurrence takes the same value as the first occurrence. In either case, each occurrence still has a distinct parameter position number.

In TimesTen mode, SQL statements containing duplicate parameters are parsed such that only distinct parameter names are considered as separate parameters. Binding is based on the position of the first occurrence of a parameter name. Subsequent occurrences of the parameter name are not given their own position numbers and all occurrences of the same parameter name take on the same value.

**Example 2–10  Duplicate parameters: Oracle mode**

To use a different value for the second occurrence of `a` in the SQL statement above in Oracle mode:

```
mycmd.setParam(1, ...); // first occurrence of :a
mycmd.setParam(2, ...); // second occurrence of :a
mycmd.setParam(3, ...); // occurrence of :b
```

To use the same value for both occurrences of `a`:

```
mycmd.setParam(1, ...); // both occurrences of :a
mycmd.setParam(3, ...); // occurrence of :b
```

Parameter `b` is considered to be in position 3 regardless, and the number of parameters is considered to be 3.

**Example 2–11  Duplicate parameters: TimesTen mode**

For the SQL statement above, in TimesTen mode the two occurrences of `a` are considered to be a single parameter, so cannot be bound separately:

```
mycmd.setParam(1, ...); // both occurrences of :a
mycmd.setParam(2, ...); // occurrence of :b
```

Note that in TimesTen mode, parameter `b` is considered to be in position 2, not position 3, and the number of parameters is considered to be 2.

**Working with REF CURSORS**

`REF CURSOR` is a PL/SQL concept, where a `REF CURSOR` is a handle to a cursor over a SQL result set and can be passed between PL/SQL and an application. In TimesTen, the cursor can be opened in PL/SQL, then the `REF CURSOR` can be passed to the application for processing. This usage is an `OUT REF CURSOR`, an `OUT parameter`
with respect to PL/SQL. As with any other OUT parameter, it must be registered using the \texttt{TTCmd::registerParam()} method. (See "Registering parameters" on page 2-9 and "Binding OUT or IN OUT parameters" on page 2-10.)

In the TimesTen implementation, the REF CURSOR is attached to a separate statement handle. The application prepares a SQL statement that has a REF CURSOR parameter on one statement handle, then, before executing the statement, binds a second statement handle as the value of the REF CURSOR. After the statement is executed, the application can describe, bind, and fetch the results using the same APIs as for any result set.

In TTClasses, because a \texttt{TTCmd} object encapsulates a single SQL statement, two \texttt{TTCmd} objects are used to support this REF CURSOR model.

Example 2–12 below demonstrates the following steps for using a REF CURSOR in TTClasses.

1. Declare a \texttt{TTCmd} object for the PL/SQL statement that returns a REF CURSOR (\texttt{cmdPLSQL} in the example).
2. Declare a \texttt{TTCmd} pointer to point to a second \texttt{TTCmd} object for the REF CURSOR (\texttt{cmdRefCursor} in the example).
3. Use the first \texttt{TTCmd} object (\texttt{cmdPLSQL}) to prepare the PL/SQL statement.
4. Use the \texttt{TTCmd::registerParam()} method of the first \texttt{TTCmd} object to register the REF CURSOR as an OUT parameter.
5. Use the first \texttt{TTCmd} object to execute the statement.
6. Use the \texttt{TTCmd::getParam()} method of the first \texttt{TTCmd} object to retrieve the REF CURSOR into the second \texttt{TTCmd} object (using \texttt{&cmdRefCursor}). There is a \texttt{getParam(int paramNo, TTCmd** rcCmd)} signature for REF CURSORS.
7. Fetch the results from the \texttt{TTCmd} object for the REF CURSOR and process as desired.
8. Drop the first \texttt{TTCmd} object.
9. Drop the pointer to the \texttt{TTCmd} object for the REF CURSOR.
10. Issue a delete statement to delete the \texttt{TTCmd} object for the REF CURSOR.

Example 2–12 Using a REF CURSOR

This example retrieves and processes a REF CURSOR from a PL/SQL anonymous block. See the preceding steps for an explanation.

---

Important:

- For passing REF CURSORS between PL/SQL and an application, TimesTen supports only OUT REF CURSORS, from PL/SQL to the application, and supports a statement returning only a single REF CURSOR.
- As noted in "Considerations when using an ODBC driver manager" on page 1-5, REF CURSOR functionality does not work in TTClasses when you use an ODBC driver manager. (This restriction does not apply to the demo \texttt{ttdm} driver manager supplied with TimesTen Quick Start.)
Managing TimesTen data

```cpp
TTCmd cmdPLSQL;
TTCmd* cmdRefCur;
TTConnection conn;
...

// c1 is a TT_INTEGER column.
cmdPLSQL.Prepare(&conn, "begin open :rc for select c1 from t; end;")
cmdPLSQL.registerParam(1, TTCmd::PARAM_OUT, SQL_REFCURSOR);
cmdPLSQL.Execute();
if (cmdPLSQL.getParam(1, &cmdRefCur) == false)
{
    SQLINTEGER fetchval;
    while (!cmdRefCursor->FetchNext()) {
        cmdRefCur->getColumn(1, &fetchval);
    }
    cmdRefCursor->Drop();
    delete cmdRefCursor;
}

cmdPLSQL.Drop();
```

Notes:
- Any TTCmd object, including one for a REF CURSOR, has an ODBC statement handle allocated for it. The REF CURSOR statement handle is dropped at the time of the Drop() statement and the resource is freed after the delete statement.
- Unlike TTCmd::getParam() calls for other data types, a getParam() call with a TTCmd** parameter for a REF CURSOR can only be called once. Subsequent calls will return NULL.

Working with rowids

Each row in a TimesTen database table has a unique identifier known as its rowid. An application can retrieve the rowid of a row from the ROWID pseudocolumn. Rowids can be represented in either binary or character format.

An application can specify literal rowid values in SQL statements, such as in WHERE clauses, as CHAR constants enclosed in single quotes.

The ODBC SQL type SQL_ROWID corresponds to the SQL type ROWID.

For parameters and result set columns, rowids are convertible to and from the C types SQL_C_BINARY, SQL_C_WCHAR, and SQL_C_CHAR. SQL_C_CHAR is the default C type for rowids. The size of a rowid is 12 bytes as SQL_C_BINARY, 18 bytes as SQL_C_CHAR, and 36 bytes as SQL_C_WCHAR.

Note that TTClasses has always supported rowids as character strings; however, beginning in Release 11.2.1.6.0, a TTClasses application can pass a rowid to a PL/SQL anonymous block as a ROWID type instead of a string. This involves using the TTCmd::registerParam() method to register the rowid input parameter as SQL_ROWID type, as shown in Example 2–13.

Example 2–13 Using a rowid

```cpp
...
TTConnection conn;
```
Using TTClasses logging

TTCmd cmd;

    cmd.Prepare(conn, "begin delete from t1 where rowid = :x; end;";
    cmd.registerParam(1, TTCmd::PARAM_IN, SQL_ROWID);
    cmd.setParameter(1, rowid_string);
    cmd.Execute();

    ...

Refer to “ROWID data type” and “ROWID specification” in Oracle TimesTen In-Memory Database SQL Reference for additional information about rowids and the ROWID data type, including usage and life.

---

Setting a timeout or threshold for executing SQL statements

TimesTen offers two ways for you to limit the time for SQL statements or procedure calls to execute, by setting either a timeout value or a threshold value. For the former, if the timeout duration is reached, the statement stops executing and an error is thrown. For the latter, if the threshold is reached, an SNMP trap is thrown but execution continues.

The query timeout limit has effect only when a SQL statement is actively executing. A timeout does not occur during commit or rollback.

Use the TTCmd methods setQueryTimeout() and setQueryThreshold() to specify these settings. There is also a getQueryThreshold() method to read the current threshold setting.

In TTClasses, these features can be operated only at the statement level, not the connection level.

For related information, see “Setting a timeout or threshold for executing SQL statements” in Oracle TimesTen In-Memory Database C Developer’s Guide.

---

Using TTClasses logging

TTClasses has a logging facility that allows applications to capture debugging information. TTClasses logging is associated with processes. You can enable logging for a specific process and produce a single output log stream for the process.

TTClasses supports different levels of logging information. See Example 2–15 on page 2-18 for more information about what is printed at each log level.

Log level WARN is very useful while developing a TTClasses application. It can also be appropriate for production applications because in this log level database query plans are generated.

Note that at the more verbose log levels (INFO and DEBUG), so much log data is generated that application performance can be adversely affected. We strongly discourage using these log levels in a production environment.

Although TTClasses logging can print to either stdout or stderr, the best approach is to write directly to a TTClasses log file. Example 2-14 demonstrates how to print TTClasses log information at log level WARN into the /tmp/ttclasses.log output file.

---

Note: Oracle TimesTen In-Memory Database does not support the PL/SQL type UROWID.
Using TTClasses XLA

Example 2–14 Printing TTClasses log information

```cpp
ofstream output;
output.open("/tmp/ttclasses.log");
TTGlobal::setLogStream(output);
TTGlobal::setLogLevel(TTLog::TTLOG_WARN);
```

First-time users of TTClasses should spend a little time experimenting with TTClasses logging to see how errors are printed at log level ERROR and how much information is generated at log levels INFO and DEBUG.

See “TTGlobal” on page 3-1 for more information about using the TTGlobal class for logging.

Using TTClasses XLA

The Transaction Log API (XLA) is a set of functions that enable you to implement applications that monitor TimesTen for changes to specified database tables and receive real-time notification of these changes.

One of the purposes of XLA is to provide a high-performance, asynchronous alternative to triggers.

XLA returns notification of changes to specific tables in the database, as well as information about the transaction boundaries for those database changes. This section shows how to acknowledge updates only at transaction boundaries (a common requirement for XLA applications), using one example that does not use and one example that does use transaction boundaries.

This section covers the following topics:

- Acknowledging XLA updates without using transaction boundaries
- Acknowledging XLA updates at transaction boundaries
- Access control impact on XLA

For additional information about XLA, see “XLA and TimesTen Event Management” in Oracle TimesTen In-Memory Database C Developer’s Guide. In addition, the TTClasses Quick Start demos include XLA demos. See ’About the TimesTen TTClasses demos’ on page 1-6.

**Important:** As noted in “Considerations when using an ODBC driver manager” on page 1-5, XLA functionality does not work when you use an ODBC driver manager.

Acknowledging XLA updates without using transaction boundaries

Example 2-15 shows basic usage of XLA, without using transaction boundaries.

Inside the HandleChange() method, depending on whether the record is an insert, update, or delete, the appropriate method from among the following is called: HandleInsert(), HandleUpdate(), or HandleDelete().

It is inside HandleChange() that you can access the flag that indicates whether the XLA record is the last record in a particular transaction. Thus there is no way in the Example 2-15 loop for the HandleChange() method to pass the information about
the transaction boundary to the loop, so that this information can influence when to call \texttt{conn.ackUpdates()}. This is not an issue under typical circumstances of only a few records per transaction. Usually only a few records are returned when you ask XLA to return at most 1000 records with a \texttt{fetchUpdatesWait} call. XLA returns records as quickly as it can, and even if huge numbers of transactions are occurring in the database, you usually can pull the XLA records out quickly, a few at a time, and XLA usually makes sure that the last record returned is on a transaction boundary. For example, if you ask for 1000 records from XLA but only 15 are returned, it is highly probable that the 15th record is at the end of a transaction.

XLA guarantees one of the following:

- A batch of records will end with a completed transaction (perhaps multiple transactions in a single batch of XLA records).

Or:

- A batch of records will contain a partial transaction, with no completed transactions in the same batch, and subsequent batches of XLA records will be returned for that single transaction until its transaction boundary has been reached.

\textit{Example 2–15  TTClasses XLA program}

This example shows a typical main loop of a TTClasses XLA program.

\begin{verbatim}
TTXlaPersistConnection conn; // XLA connection
TTXlaVariableList list(&conn); // tables being monitored
TTXlaUpdateDesc_t ** arry; // ptr to returned XLA recs
int records_fetched;
// ...
loop {
  // fetch the updates
  conn.fetchUpdatesWait(&arry, MAX_RECS_TO_FETCH, &records_fetched, ...);
  // Interpret the updates
  for(j=0;j < records_fetched;j++) {
    TTXlaUpdateDesc_t *p;
    p = arry[j];
    list.HandleChange(p, NULL);
  } // end for each record fetched
  // periodically call ackUpdates()
  if (/* some condition is reached */) {
    conn.ackUpdates();
  }
} // loop
\end{verbatim}

\textbf{Acknowledging XLA updates at transaction boundaries}

XLA applications should verify whether the last record in a batch of XLA records has a transaction boundary, and call \texttt{ackUpdates()} only on transaction boundaries. This way, when the application or system or database fails, the XLA bookmark is at the start of a transaction after the system recovers. This is especially important when operations involve a large number of rows. If a bulk insert, update, or delete operation has been performed on the database and the XLA application asks for 1000 records, it may or may not receive all 1000 records. The last record returned through XLA will
probably not have the end-of-transaction flag. In fact, if the transaction has made changes to 10,000 records, then clearly a minimum of 10 blocks of 1000 XLA records must be fetched before reaching the transaction boundary.

Calling ackUpdates() for every transaction boundary is not recommended, however, because ackUpdates() is a relatively expensive operation. Users need to balance overall system throughput with recovery time and disk space requirements. (Recall that a TimesTen transaction log file cannot be deleted by a checkpoint operation if XLA has a bookmark that references that log file. See “ttiLogHolds” in Oracle TimesTen In-Memory Database Reference for related information.) Depending on system throughput, recovery time, and disk space requirements, some applications may find it appropriate to call ackUpdates() once or several times per minute, while other applications may need only call it once or several times per hour.

The HandleChange() method has a second parameter to allow passing information between HandleChange() and the main XLA loop. Compare Example 2–15 with Example 2–16, specifically the do_acknowledge setting and the &do_acknowledge parameter of the HandleChange() call.

**Example 2–15** TTClasses XLA program using transaction boundaries

In this example, ackUpdates() is called only when the do_acknowledge flag indicates that this batch of XLA records is at a transaction boundary.

```
TTXlaPersistConnection conn; // XLA connection
TTXlaTableList list(&conn); // tables being monitored
TTXlaUpdateDesc_t ** arry; // ptr to returned XLA recs
int records_fetched;
int do_acknowledge;
int j;

// ... loop {
    // fetch the updates
    conn.fetchUpdatesWait(arry, MAX_RECS_TO_FETCH, &records_fetched,...);
    do_acknowledge = FALSE;

    // Interpret the updates
    for(j=0; j < records_fetched; j++){
        TTXlaUpdateDesc_t *p;
        p = arry[j];
        list.HandleChange(p, &do_acknowledge);
    }

    // periodically call ackUpdates()
    if (do_acknowledge == TRUE) /* and some other conditions ... */) {
        conn.ackUpdates();
    }
} // loop
```

In addition to this change to the XLA main loop, the HandleChange() method must be overloaded to have two parameters (TTXlaUpdateDesc_t*, void* pData). See “HandleChange()” on page 3-51. Note that the Quick Start xlasubscriber1 demo code shows the use of a pData parameter. (See “About the TimesTen TTClasses demos” on page 1-6.)
Access control impact on XLA

“Considering TimesTen features for access control” on page 2-4 provides a brief overview of how TimesTen access control affects operations in the database. Access control includes impact on XLA.

Any XLA functionality requires the system privilege XLA. This includes connecting to TimesTen as an XLA reader, executing XLA-related TimesTen C functions, and executing XLA-related TimesTen built-in functions.

You can refer to “Access control impact on XLA” in Oracle TimesTen In-Memory Database C Developer’s Guide for additional details.

Note: A user with the XLA privilege can be notified of any DML statement that executes in the database. As a result, the user with XLA privilege can obtain information about database objects that he or she has not otherwise been granted access to. In practical terms, the XLA privilege is effectively the same as the SELECT ANY TABLE privilege.
This reference chapter contains descriptions of TTClasses external classes and their methods. It is divided into the following sections:

- Commonly used TTClasses
- System catalog classes
- XLA classes

**Note:** Most methods documented in this chapter also support a signature with a `TTStatus&` parameter at the end of the parameter list. This is for situations when you want to suppress exceptions for the method call and instead process the `TTStatus` object manually for errors. These signatures are not specifically documented, however, because this is not a typical mode of use. For additional information and an example, see the Usage section under “TTStatus” on page 3-3.

### Commonly used TTClasses

This section discusses the following classes:

- TTGlobal
- TTStatus
- TTConnection
- TTConnectionPool
- TTCmd

#### TTGlobal

The `TTGlobal` class provides a logging facility within TTClasses.

**Usage**

The `TTGlobal` logging facility can be very useful for debugging problems inside a TTClasses program. Note, however, that the most verbose logging levels (`TTLog::TTLOG_INFO` and `TTLog::TTLOG_DBG`) can generate an extremely large amount of output. Use these logging levels during development or when trying to diagnose a bug instead of during production.

When logging from a multithreaded program, you may encounter a problem where log output from different program threads is intermingled when written to disk. To alleviate this problem, disable `ostream` buffering with the `ios_base::unitbuf` I/O
stream manipulator, as in the following example, which sends TTClasses logging to the app_log.txt file at logging level TTLog::TTLOG_ERR.

```cpp
ofstream log_file("app_log.txt");
log_file << std::ios_base::unitbuf;
TTGlobal::setLogStream(log_file);
TTGlobal::setLogLevel(TTLog::TTLOG_ERR);
```

See “Using TTClasses logging” on page 2-16 for more information about using TTGlobal.

**Public members**

None.

**Public methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setLogStream()</code></td>
<td>Specifies where TTClasses logging information should be sent.</td>
</tr>
<tr>
<td><code>setLogLevel()</code></td>
<td>Specifies the verbosity level of TTClasses logging.</td>
</tr>
<tr>
<td><code>disableLogging()</code></td>
<td>Disables TTClasses logging.</td>
</tr>
<tr>
<td><code>sqlhenv()</code></td>
<td>Returns the underlying ODBC environment object (type SQLHENV).</td>
</tr>
</tbody>
</table>

`setLogStream()`

```cpp
static void setLogStream(ostream& stream)
```

Specifies where TTClasses logging information should be sent. By default, if TTClasses logging is enabled, logging is to stderr. Using this method, an application can specify logging to a file (or any other `ostream&`), such as in the following example that sets logging to `app_log.txt`:

```cpp
ofstream log_file("app_log.txt");
TTGlobal::setLogStream(log_file);
```

`setLogLevel()`

```cpp
static void setLogLevel(TTLog::TTLOG_LEVEL level)
```

This method specifies the verbosity level of TTClasses logging. Table 3–1 describes TTClasses logging levels. The levels are cumulative.

<table>
<thead>
<tr>
<th>Logging level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTLog::TTLOG_NIL</td>
<td>No logging.</td>
</tr>
<tr>
<td>TTLog::TTLOG_FATAL_ERR</td>
<td>Logs fatal errors (serious misuse of TTClasses methods).</td>
</tr>
<tr>
<td>TTLog::TTLOG_ERR</td>
<td>Logs all errors, such as SQL_ERROR return codes.</td>
</tr>
<tr>
<td>TTLog::TTLOG_WARN</td>
<td>(Default) Also logs warnings and all calls to TTCmd::Prepare(), including the SQL string being prepared. Prints all database optimizer query plans.</td>
</tr>
<tr>
<td>TTLog::TTLOG_INFO</td>
<td>Also logs informational messages, such as calls to most methods on TTCmd and TTConnection objects, including the SQL string where appropriate.</td>
</tr>
</tbody>
</table>

Table 3–1 TTClasses logging levels
To set the logging level to TTLog::TTLOG_ERR, for example, add the following line to your program:

```cpp
TTGlobal::setLogLevel(TTLog::TTLOG_ERR);
```

This method disables all TTClasses logging. Note that the following two statements are identical:

```cpp
TTGlobal::disableLogging();
TTGlobal::setLogLevel(TTLog::TTLOG_NIL);
```

Retrieves the underlying ODBC environment object.

**TTStatus**

The TTStatus class is used by other classes in the TTClasses library to catch error and warning exceptions. You can think of TTStatus as a value-added C++ wrapper around the SQLError ODBC function.

**Usage**

By default, TTStatus objects are thrown as exceptions whenever an error occurs. This allows C++ applications to use `{try/catch}` blocks to detect and recover from database errors.

**Example 3–1** shows typical use of TTStatus.

**Example 3–1 Exception handling**

```cpp
... TTCmd myCmd;
try {
    myCmd.ExecuteImmediate(&conn, "create table dummy (c1 int)");
} catch (TTStatus st) {
    cerr << "Error creating table: " << st << endl;
    // Rollback, exit(), throw -- whatever is appropriate
}
```

Another supported (but not typical) mode of use for TTStatus is to selectively suppress exceptions and allow the application to manually check a TTStatus object for error conditions. You can use this mode for a particular method call by initializing a TTStatus object with the value TTStatus::DO_NOT_THROW, then passing that object as the last parameter of a method call. Most TTClasses methods documented in Table 3–1 (Cont.) TTClasses logging levels

<table>
<thead>
<tr>
<th>Logging level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTLog::TTLOG_DEBUG</td>
<td>Also logs debugging information, such as all bound parameter values for each call to TTCmd::Execute().</td>
</tr>
</tbody>
</table>

Table 3–1 (Cont.) TTClasses logging levels

<table>
<thead>
<tr>
<th>Logging level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTLog::TTLOG_ERR</td>
<td>Also logs debugging information, such as all bound parameter values for each call to TTCmd::Execute().</td>
</tr>
</tbody>
</table>
this chapter also support a signature with this TTStatus& parameter as the last parameter in the calling sequence.

Example 3-2 shows this usage:

**Example 3-2  Suppressing exceptions**

... 
TTCmd myCmd;
TTStatus myStat(TTStatus::DO_NOT_THROW);
myCmd.ExecuteImmediate(conn, "create table dummy (c1 int)", myStat);
if (myStat.rc == SQL_ERROR)
{
    // handle the error
}
...

Subclasses

TTStatus has the following subclasses:

- TTError
- TTWarning

**TTError**

TTError is a subclass of TTStatus and is used to encapsulate ODBC errors (return codes SQL_ERROR and SQL_INVALID_HANDLE).

**TTWarning**

TTWarning is a subclass of TTStatus and is used to encapsulate ODBC warnings (return code SQL_SUCCESS_WITH_INFO).

ODBC warnings are usually not as serious as ODBC errors and should be handled with different logic. Simply logging ODBC warnings is usually appropriate, but ODBC errors should typically be handled programatically.

Example 3-3 shows usage of the TTError and TTWarning subclasses.

**Example 3-3  Exception handling, distinguishing between errors and warnings**

This example shows the use of TTError and TTWarning. TTError objects are thrown for ODBC errors. TTWarning objects are thrown for ODBC warnings.

```cpp
// catching TTError & TTWarning exceptions
try {
    // some TTClasses method calls
}
catch (TTWarning warn) {
    cerr << "Warning encountered: " << warn << endl;
}
catch (TTError err) {
    // handle the error; this could be a serious problem
}
```

3-4  Oracle TimesTen In-Memory Database TTClasses Guide
Commonly used TTClasses

Class Descriptions

Public members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rc</td>
<td>Return code from the failing ODBC call. Possible values for this field are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA_FOUND, and SQL_INVALID_HANDLE.</td>
</tr>
<tr>
<td>native_error</td>
<td>TimesTen native error number (if any) for the failing ODBC call.</td>
</tr>
<tr>
<td>odbc_error</td>
<td>ODBC error state for the failing ODBC call.</td>
</tr>
<tr>
<td>err_msg</td>
<td>ASCII printable error message for the failing ODBC call.</td>
</tr>
<tr>
<td>TTSTATUS_ENUM</td>
<td>Use the value TTStatus::DO_NOT_THROW to initialize a TTStatus object if you want to suppress exceptions for a method call. See Example 3–2 on page 3-4.</td>
</tr>
</tbody>
</table>

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostream()</td>
<td>Prints errors to a stream.</td>
</tr>
<tr>
<td>resetErrors()</td>
<td>Resets the TTStatus object or just the rc value, as specified.</td>
</tr>
<tr>
<td>isConnectionInvalid()</td>
<td>Indicates whether the database connection is invalid.</td>
</tr>
<tr>
<td>throwError()</td>
<td>Throws an error from the TTStatus object (not typical use).</td>
</tr>
</tbody>
</table>

friend ostream operator<<(ostream &stat)

This method can be used to print the error to a stream.

void resetErrors(bool reset_all=false)

Use this method to reset a TTStatus object (relevant only when using method calls with TTStatus parameters). Use a value of TRUE to completely reset the TTStatus object, or FALSE (default) to reset only the rc value.

bool isConnectionInvalid() const

Returns TRUE if the database connection is invalid, or FALSE if it is valid. Specifically, "invalid" refers to situations when a TimesTen error 846 or 994 is encountered. See "Errors 0 - 999" in Oracle TimesTen In-Memory Database Error Messages and SNMP Traps for information about those errors.

void throwError()

Assuming exceptions are enabled (see "Usage" on page 3-3), this is an alternative, but not typical, way to throw an exception. In most cases the following two blocks of code are equivalent, but the former is more typical:

try {
// ...

if (/* something has gone wrong */)  
    throw stat;
}  
  catch (TTStatus st) {  
    cerr << "Caught exception: " << st << endl;  
  }  

Or:

try {  
  // ...  
  if (/* something has gone wrong */)  
    stat.throwError();  
  }  
  catch (TTStatus st) {  
    cerr << "Caught exception: " << st << endl;  
  }

TTConnection

The TTConnection class encapsulates the concept of a connection to a TimesTen database. You can think of TTConnection as a value-added C++ wrapper around the ODBC connection (HDBC) handle.

Usage

All applications that use TimesTen must create at least one TTConnection object. Multithreaded applications that wish to use TimesTen from multiple threads simultaneously must create more than one TTConnection object. Use one of the following strategies:

- Create one TTConnection object for each thread when the thread is created.
- Create a pool of TTConnection objects when the application process starts. They are shared by the threads in the process. See "TTConnectionPool" on page 3-11 for additional information about this option.

A TimesTen connection cannot be inherited from a parent process. If a process opens a database connection before creating a child process, the child cannot use the same connection. A child’s attempt to use a parent’s database connection will likely cause application failure or a core dump.

Applications should not frequently make and then drop database connections, because connecting and disconnecting are both relatively expensive operations. In addition, short-lived connections eliminate the benefits of prepared statements. (See "Using TTCmd, TTConnection, and TTConnectionPool" on page 2-1 for information about preparing statements.) Instead, establish database connections at the beginning of the application process and reuse them for the life of the process.

Note: If you have reason to manipulate the underlying ODBC connection object directly, use the TTConnection::getHdbc() method.

Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See "Access control for connections" on page 2-6.

Commonly used TTClasses
### Public members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVER_COMPLETION_ENUM</td>
<td>This is to specify whether there will be a prompt for the data source to connect to (also depending on whether a data source is specified in the connect string). Valid values are TTConnection::DRIVER_NOPROMPT, TTConnection::DRIVER_COMPLETE, TTConnection::DRIVER_PROMPT, and TTConnection::DRIVER_COMPLETE_REQUIRED. These correspond to the values SQL_DRIVER_NOPROMPT, SQL_DRIVER_COMPLETE, SQL_DRIVER_PROMPT, and SQL_DRIVER_COMPLETE_REQUIRED for the standard ODBC SQLDriverConnection function.</td>
</tr>
</tbody>
</table>

### Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect()</td>
<td>Opens a new connection to a TimesTen data store.</td>
</tr>
<tr>
<td>Disconnect()</td>
<td>Closes a connection to a TimesTen data store.</td>
</tr>
<tr>
<td>Commit()</td>
<td>Commits a transaction to the TimesTen data store.</td>
</tr>
<tr>
<td>Rollback()</td>
<td>Rolls back changes made to the database through this connection since the last call to Commit() or Rollback().</td>
</tr>
<tr>
<td>isConnected()</td>
<td>Returns TRUE if the object is connected to TimesTen.</td>
</tr>
<tr>
<td>getHdbc()</td>
<td>Returns the ODBC connection handle (type HDBC) associated with this connection.</td>
</tr>
<tr>
<td>SetIsoReadCommitted()</td>
<td>Sets the transaction isolation level of the connection to be TXN_READ_COMMITTED.</td>
</tr>
<tr>
<td>SetIsoSerializable()</td>
<td>Sets the transaction isolation level of the connection to be TXN_SERIALIZABLE.</td>
</tr>
<tr>
<td>DurableCommit()</td>
<td>Performs a durable commit operation on the data store.</td>
</tr>
<tr>
<td>CompactDataStore()</td>
<td>Compacts the data store by calling the ttCompact or ttCompactTS TimesTen built-in procedure, as specified.</td>
</tr>
<tr>
<td>SetLockWait()</td>
<td>Sets the lock timeout interval for the connection by calling the ttLockWait TimesTen built-in procedure.</td>
</tr>
<tr>
<td>SetPrefetchCloseOn()</td>
<td>Turns on the TT_PREFETCH_CLOSE connection option. This is useful for optimizing SELECT query performance for client/server connections to TimesTen.</td>
</tr>
<tr>
<td>SetPrefetchCloseOff()</td>
<td>Turns off the TT_PREFETCH_CLOSE connection option.</td>
</tr>
<tr>
<td>SetPrefetchCount()</td>
<td>Allows a user application to tune the number of rows that the TimesTen ODBC driver SQLFetch call will prefetch for a SELECT statement.</td>
</tr>
<tr>
<td>SetAutoCommitOff()</td>
<td>Sets AUTOCOMMIT off for the connection.</td>
</tr>
<tr>
<td>SetAutoCommitOn()</td>
<td>Sets AUTOCOMMIT on for the connection.</td>
</tr>
<tr>
<td>GetTTContext()</td>
<td>Returns the connection context value.</td>
</tr>
</tbody>
</table>

```cpp
virtual void Connect(const char* connStr) {
    // Implementation details...
}
```
virtual void Connect(const char* connStr, const char* username, const char* password) {
    // Connect with username and password
}
virtual void Connect(const char* connStr, DRIVER_COMPLETION_ENUM driverCompletion) {
    // Connect with driverCompletion
}

Opens a new connection to a TimesTen data store. The connection string specified in the connStr parameter is used to create the connection. Specify a user and password (either as part of the connect string or as separate parameters) or a DRIVER_COMPLETION_ENUM value (refer to “Public members” on page 3-7).

Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See “Access control for connections” on page 2-6.

Example 3–4 Using the Connect() method and checking for errors

A TTStatus object is thrown as an exception if an error occurs. Any exception warnings are usually informational and can often be safely ignored. The following logic is preferred for use of the Connect() method.

Note that TTWarning and TTError are subclasses of TTStatus.

```cpp
TTConnection conn;
...
try {
    conn.Connect("DSN=mydsn", "myuser", "mypassword");
} catch (TTWarning warn) {
    // warnings from Connect() are usually informational
    cerr << "Warning while connecting to TimesTen: " << warn << endl;
} catch (TTError err) {
    // handle the error; this could be a serious problem
}
```

Disconnect()

disconnect()

Closes a connection to a TimesTen data store. A TTStatus object is thrown as an exception if an error occurs.

Commit()

void Commit();

Commits a transaction to the TimesTen database. All other operations performed on this connection since the last call to the Commit() or Rollback() method will be committed. A TTStatus object is thrown as an exception if an error occurs.

Rollback()

void Rollback();

Rolls back (cancels) a transaction. Any changes made to the database through this connection since the last call to Commit() or Rollback() will be undone. A TTStatus object is thrown as an exception if an error occurs.

isConnected()

bool isConnected();
Returns TRUE if the object is connected to TimesTen (using the Connect() method) or FALSE if not.

getHdbc()

HDBC getHdbc()

Returns the ODBC connection handle associated with this connection.

SetIsoReadCommitted()

void SetIsoReadCommitted()

Sets the transaction isolation level of the connection to be TXN_READ_COMMITTED. Read-committed isolation offers the best combination of single-transaction performance and good multiconnection concurrency.

SetIsoSerializable()

void SetIsoSerializable()

Sets the transaction isolation level of the connection to be TXN_SERIALIZABLE. In general, serializable isolation offers fair individual transaction performance but extremely poor concurrency. READ_COMMITTED isolation level is preferable over SERIALIZABLE isolation level in almost all situations.

DurableCommit()

void DurableCommit()

Performs a durable commit operation on the data store. A durable commit operation flushes the in-memory transaction log buffer to disk. It calls the ttDurableCommit TimesTen built-in procedure.

See 'ttDurableCommit' in Oracle TimesTen In-Memory Database Reference.

CompactDataStore()

void CompactDataStore(int blocks)

Compacts the data store, as specified:

- For a blocks value less than or equal to zero, it compacts the permanent and temporary data partitions in their entirety by calling the ttCompact TimesTen built-in procedure.
- For a blocks value greater than zero, it compacts a portion of the data store, according to the number of blocks specified, by calling the ttCompactTE built-in procedure.

This method is supported for backward compatibility. New applications should not call it.

SetLockWait()

void SetLockWait(int secs)

Sets the lock timeout interval for the connection by calling the ttLockWait TimesTen built-in procedure with the secs parameter. In general, a two-second or three-second
lock timeout is sufficient for most applications. The default lock timeout interval is 10 seconds.

See “ttLockWait” in Oracle TimesTen In-Memory Database Reference.

SetPrefetchCloseOn()

```cpp
void SetPrefetchCloseOn()
```

Turns on the TT_PREFETCH_CLOSE connection option, which is useful for optimizing SELECT query performance for client/server connections to TimesTen. Note that this method provides no benefit for an application using a direct connection to TimesTen.

See “Bulk fetch rows of TimesTen data” in Oracle TimesTen In-Memory Database C Developer’s Guide for more information about TT_PREFETCH_CLOSE.

SetPrefetchCloseOff()

```cpp
void SetPrefetchCloseOff()
```

Turns off the TT_PREFETCH_CLOSE connection option.

SetPrefetchCount()

```cpp
void SetPrefetchCount(int numrows)
```

Allows a user application to tune the number of rows that the TimesTen ODBC driver internally fetches at a time for a SELECT statement. The value of numrows must be between 1 and 128, inclusive.

Note: This method is not equivalent to executing `TTCmd::FetchNext()` multiple times. Instead, proper use of this parameter reduces the amount of time for each call to `TTCmd::FetchNext()`.

See “Bulk fetch rows of TimesTen data” in Oracle TimesTen In-Memory Database C Developer’s Guide for more information about TT_PREFETCH_COUNT.

SetAutocommitOff()

```cpp
void SetAutoCommitOff()
```

Sets AUTOCOMMIT off for the connection. This method is automatically called by `TTConnection::Connect()`, because TimesTen runs with optimal performance only with AUTOCOMMIT turned off. Note that when AUTOCOMMIT is off, committing SELECT statements requires explicit calls to `TTCmd::Close()`.

SetAutocommitOn()

```cpp
void SetAutoCommitOn()
```

Sets AUTOCOMMIT on for the connection, which means that every SQL statement occurs in its own transaction. This is generally not advisable, because TimesTen runs much faster with AUTOCOMMIT turned off.
Commonly used TTClasses

### GetTTContext()

```c
void GetTTContext(char* output)
```

Returns the context value of the connection, a value that is unique for each connection to a TimesTen data store. The context of a connection can be used to correlate TimesTen connections with PIDs using the `ttStatus` TimesTen utility, for example.

The context value is returned through the `output` parameter, which requires an array of CHAR[17] or larger.

This method calls the `ttContext` TimesTen built-in procedure. See “ttContext” in Oracle TimesTen In-Memory Database Reference.

### TTConnectionPool

The `TTConnectionPool` class is used by multithreaded applications to manage a pool of connections.

In general, multithreaded applications can be written using one of two basic strategies:

- If there is a relatively small number of threads and the threads are long-lived, each thread can be assigned to a different connection, which is used for the duration of the application. In this scenario, the `TTConnectionPool` class is not necessary.

- If there is a large number of threads in the process, or if the threads are short-lived, a pool of idle connections can be established. These connections are used for the duration of the application. When a thread needs to perform a database transaction, it checks out an idle connection from the pool, performs its transaction, then returns the connection to the pool. This is the scenario that the `TTConnectionPool` class assists with.

**Note:** For best overall performance, TimesTen recommends having one or two concurrent direct connections to the database for each CPU of the database server. For no reason should your number of concurrent direct connections (the size of your connection pool) be more than twice the number of CPUs on the database server. In client/server mode, however, TimesTen supports many more connections per CPU efficiently.

### Usage

To use the `TTConnectionPool` class, an application creates a single instance of the class. It then creates a number of `TTConnection` objects, instances of either the `TTConnection` class or a user class that extends it, but does not call their `Connect()` methods directly. Instead, the application uses the `TTConnectionPool::AddConnectionToPool()` method to place connection objects into the pool, then calls `TTConnectionPool::ConnectAll()` to establish all the connections to TimesTen. In the background, `ConnectAll()` loops through all the `TTConnection` objects to call their `Connect()` methods.

Threads for TimesTen applications use the `getConnection()` and `freeConnection()` methods to get and return idle connections.
Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See “Access control for connections” on page 2-6.

Public members
None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddConnectionToPool()</td>
<td>Adds a TTConnection object (possibly an object of a class derived from TTConnection) to the connection pool.</td>
</tr>
<tr>
<td>ConnectAll()</td>
<td>Connects all the TTConnection objects to TimesTen simultaneously.</td>
</tr>
<tr>
<td>getConnection()</td>
<td>Checks out an idle connection from the connection pool for a thread.</td>
</tr>
<tr>
<td>freeConnection()</td>
<td>Returns a connection to the pool for reassignment to another thread.</td>
</tr>
<tr>
<td>DisconnectAll()</td>
<td>Disconnects all connections in the connection pool from TimesTen.</td>
</tr>
<tr>
<td>getStats()</td>
<td>Queries the TTConnectionPool object for status information.</td>
</tr>
</tbody>
</table>

**Important:** If you wish to use TTConnectionPool and extend TTConnection, do not override the TTConnection::Connect() method that has *driverCompletion* in the calling sequence, because there is no corresponding TTConnectionPool::ConnectAll() method. Instead, override either of the following Connect() methods:

```c++
virtual void Connect(const char* connStr)
virtual void Connect(const char* connStr, const char* username, const char* password)
```

Then use the appropriate corresponding ConnectAll() method.

**Method Description**

- **AddConnectionToPool()**
  
  This method is used to add a TTConnection object (possibly an object of a class derived from TTConnection) to the connection pool. It returns -1 if there is an error.

- **ConnectAll()**
  
  After TTConnection objects have been added to the connection pool by AddConnectionToPool(), the ConnectAll() method can be used to connect all of the TTConnection objects to TimesTen simultaneously. The connection string specified in the connStr parameter is used to create the connection. Specify a user and password, either as part of the connect string or as separate parameters.

  A TTStatus object is thrown as an exception if an error occurs.
Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See "Access control for connections" on page 2-6.

**getConnection()**

```cpp
TTConnection* getConnection(int timeout_millis=0)
```

Checks out an idle connection from the connection pool for use by a thread. A pointer to an idle TTConnection object is returned. The thread should then perform a transaction, ending with either Commit() or Rollback(), and then should return the connection to the pool using the freeConnection() method.

If no idle connections are in the pool, the thread calling getConnection() will block until a connection is returned to the pool by a call to freeConnection(). An optional timeout, in milliseconds, can be provided. If this is provided, getConnection() waits for a free connection for no more than `timeout_millis` milliseconds. If no connection is available in that time then getConnection() returns NULL to the caller.

**freeConnection()**

```cpp
void freeConnection(TTConnection* connP)
```

Returns a connection to the pool for reassignment to another thread. Applications should not free connections that are in the midst of a transaction. TTConnection::Commit() or Rollback() should be called immediately prior to freeConnection().

**DisconnectAll()**

```cpp
void DisconnectAll()
```

Disconnects all connections in the connection pool from TimesTen. Applications must call DisconnectAll() prior to termination in order to avoid overhead associated with process failure analysis and recovery. A TTStatus object is thrown as an exception if an error occurs.

**getStats()**

```cpp
void getStats(int* nGets, int* nFrees, int* nWaits, int* nTimeouts,
              int* maxInUse, int* nForcedCommits)
```

Queries the TTConnectionPool for status information. The following data is returned:

- **nGets**: Number of calls to getConnection().
- **nFrees**: Number of calls to freeConnection().
- **nWaits**: Number of times a call to getConnection() had to wait before returning a connection.
- **nTimeouts**: Number of calls to getConnection() that timed out.
- **maxInUse**: High point for the most number of connections in use at one time.
- **nForcedCommits**: Number of times that freeConnection() had to call Commit() on a connection before checking it into the pool. If this counter is non-zero, the user application is not calling TTConnection::Commit() or Rollback() before returning a connection to the pool.
**TTCmd**

A TTCmd object encapsulates a single SQL statement that will be used multiple times in an application program. You can think of TTCmd as a value-added C++ wrapper around the ODBC statement (HSTMT) handle.

TTCmd has three categories of public methods:

- Public methods for general use and non-batch operations
- Public methods for obtaining TTCmd object properties
- Public methods for batch operations

**Important:** A number of TTCmd methods return an error when you use an ODBC driver manager. See "Considerations when using an ODBC driver manager" on page 1-5 for a list.

**Usage**

Each SQL statement executed multiple times in a program should have its own TTCmd object. Each of these TTCmd objects should be prepared once during program initialization, then executed with the Execute() method multiple times as the program runs.

Only database operations that are to be executed a small number of times should use the ExecuteImmediate() method. Note that ExecuteImmediate() is not compatible with any type of SELECT statement. All queries must use Prepare() plus Execute() instead. ExecuteImmediate() is also incompatible with INSERT, UPDATE, or DELETE statements that are subsequently polled using getRowcount() to see how many rows were inserted, updated or deleted. These limitations have been placed on ExecuteImmediate() to discourage its use except in a few particular situations (for example, for creating or dropping a table).

**Note:** If you have reason to manipulate the underlying ODBC statement object directly, use the TTCmd::getHandle() method.

Note that TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. See “Considering TimesTen features for access control” on page 2-4.

**Public members**

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTMOD_PARAM_INPUTOUTPUT_TYPE</td>
<td>This is used to specify whether a parameter is IN, OUT, or IN OUT when registering the parameter. Supported values are PARAM_IN, PARAM_INOUT, and PARAM_OUT; see &quot;Registering parameters&quot; on page 2-9.</td>
</tr>
</tbody>
</table>

**Public methods for general use and non-batch operations**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare()</td>
<td>Associates a SQL statement with the TTCmd object.</td>
</tr>
</tbody>
</table>
### Commonly used TTClasses

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RePrepare()</td>
<td>Allows a statement to be reprepared.</td>
</tr>
<tr>
<td>Execute()</td>
<td>Invokes a SQL statement that has been prepared for execution.</td>
</tr>
<tr>
<td>ExecuteImmediate()</td>
<td>Invokes a SQL statement that has not been previously prepared.</td>
</tr>
<tr>
<td>FetchNext()</td>
<td>Fetches rows from the result set, one at a time. It returns 0 when a row was successfully fetched, and 1 when no more rows are available.</td>
</tr>
<tr>
<td>Close()</td>
<td>Closes the result set when the application has finished fetching rows.</td>
</tr>
<tr>
<td>Drop()</td>
<td>Frees a prepared SQL statement and all resources associated with it.</td>
</tr>
<tr>
<td>getHandle()</td>
<td>Retrieves the underlying ODBC statement handle.</td>
</tr>
<tr>
<td>setQueryTimeout()</td>
<td>Sets a timeout value for SQL statements.</td>
</tr>
<tr>
<td>setQueryThreshold()</td>
<td>Sets a threshold time limit for execution of each SQL statement. If it is exceeded, a warning is written to the support log and an SNMP trap is thrown.</td>
</tr>
<tr>
<td>getMaxRows()</td>
<td>Sets a limit on the number of rows returned by a SELECT statement.</td>
</tr>
<tr>
<td>getRowCount()</td>
<td>Returns the number of rows that were affected by the recently executed SQL operation.</td>
</tr>
<tr>
<td>registerParam()</td>
<td>Registers a parameter for binding. This is required for OUT or IN OUT parameters.</td>
</tr>
<tr>
<td>setParam()</td>
<td>Each call sets the value of a specified parameter before executing a prepared SQL statement.</td>
</tr>
<tr>
<td>setParamLength()</td>
<td>Sets the length, in bytes, of the specified input parameter.</td>
</tr>
<tr>
<td>setParamNull()</td>
<td>Sets the value of a parameter to NULL before executing a prepared SQL statement.</td>
</tr>
<tr>
<td>getParam()</td>
<td>Each call gets the output value of a specified OUT or IN OUT parameter after executing a prepared SQL statement.</td>
</tr>
<tr>
<td>getColumn()</td>
<td>Retrieves the value in the specified column of the current row of the result set.</td>
</tr>
<tr>
<td>getColumnNullable()</td>
<td>Retrieves the value in the specified column of the current row of the result set and returns a boolean to indicate whether the value is NULL.</td>
</tr>
<tr>
<td>getNextColumn()</td>
<td>Retrieves the value in the next column of the current row of the result set.</td>
</tr>
<tr>
<td>getNextColumnNullable()</td>
<td>Retrieves the value in the next column of the current row of the result set and returns a boolean to indicate whether the value is NULL.</td>
</tr>
<tr>
<td>isColumnNull()</td>
<td>Indicates whether the value in the specified column of the current row is NULL.</td>
</tr>
<tr>
<td>getColumnLength()</td>
<td>Returns the length of the specified column, in bytes.</td>
</tr>
<tr>
<td>printColumn()</td>
<td>Prints the value in the specified column of the current row to an output stream.</td>
</tr>
</tbody>
</table>

Class Descriptions  3-15
Prepare()

```c
void Prepare(TTConnection* cP, const char* sqlp)
```

This method associates a SQL statement with the TTCmd object. It takes two parameters:
- A pointer to a TTConnection object, which should be already connected to the data store by a call to TTConnection::Connect().
- A const char* parameter for the SQL statement being prepared.

Note that TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time. See "Considering TimesTen features for access control" on page 2-4.

RePrepare()

```c
void RePrepare(TTConnection* cP)
```

This method allows a statement to be reprepared. It is useful only when a statement handle in a prepared statement has been invalidated.

Execute()

```c
void Execute()
```

This method invokes a SQL statement that has been prepared for execution with the Prepare() method, after any necessary parameter values are defined using setParam() calls.

If the SQL statement is a SELECT statement, this method executes the query but does not return any rows from the result set. Use the FetchNext() method to fetch rows from the result set one at a time. Use the Close() method to close the result set when all appropriate rows have been fetched. For SQL statements other than SELECT, no cursor is opened, and the Close() method does not need to be called.

A TTStatus object is thrown as an exception if an error occurs.

Note that TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time. See "Considering TimesTen features for access control" on page 2-4.

ExecuteImmediate()

```c
int ExecuteImmediate(TTConnection* cP, const char* sqlp)
```

This method invokes a SQL statement that has not been previously prepared.
ExecuteImmediate() is a convenient alternative to using Prepare() and Execute() when a SQL statement is to be executed only a small number of times. Use ExecuteImmediate() for DDL statements such as CREATE TABLE and DROP TABLE, and infrequently used DML statements that do not return a result set (for example, DELETE FROM table_name).

ExecuteImmediate() is incompatible with SQL statements that return a result set. In addition, statements executed through ExecuteImmediate() cannot subsequently be queried by getRowCount() to get the number of rows affected by a DML operation. Because of this, ExecuteImmediate() calls getRowCount() automatically, and its value is the integer return value of this method.

A TTStatus object is thrown as an exception if an error occurs.

FetchNext()

int FetchNext()

After executing a prepared SQL SELECT statement using the Execute() method, use the FetchNext() method to fetch rows from the result set, one at a time.

After fetching a row of the result set, use one of the overloaded versions of the getColumn() method to fetch values from the current row.

If no more rows remain in the result set, FetchNext() returns 1. If a row is returned, FetchNext() returns 0.

After executing a SELECT statement using the Execute() method, the result set must be closed using the Close() method after all desired rows have been fetched. Note that after the Close() method is called, the FetchNext() method cannot be used to fetch additional rows from the result set.

A TTStatus object is thrown as an exception if an error occurs.

Close()

void Close()

If a SQL SELECT statement is executed using the Execute() method, a cursor is opened which may be used to fetch rows from the result set. When the application is finished fetching rows from the result set, it must be closed with the Close() method.

Failure to close the result set may result in locks being held on rows for too long, causing concurrency problems as well as memory leaks and other errors.

A TTStatus object is thrown as an exception if an error occurs.

Drop()

void Drop()

If a prepared SQL statement will not be used in the future, the statement and resources associated with it can be freed by calling the Drop() method. The TTCmd object may be reused for another statement by calling Prepare() again.

It is more efficient to use multiple TTCmd objects to execute multiple SQL statements. Use the Drop() method only if it is certain that a particular SQL statement will not be used again.

A TTStatus object is thrown as an exception if an error occurs.
**Commonly used TTClasses**

```c
SQLHSTMT getHandle()
```

If you have a need to manipulate the underlying ODBC statement object, use this method to retrieve the statement handle.

```c
void setQueryTimeout(const int nSecs)
```

Use this method to specify how long, in seconds, any SQL statement (not just a query) will execute before timing out.

This has the same effect as using `SQLSetStmtOption` to set SQL_QUERY_TIMEOUT or setting the TimesTen connection attribute `SqlQueryTimeout`. (See “Setting a timeout value for SQL statements” in Oracle TimesTen In-Memory Database C Developer’s Guide.)

By default there is no timeout.

```c
void setQueryThreshold(const int nSecs)
```

Use this method to specify a threshold time limit, in seconds, for SQL statements (not just queries). If the execution time of a statement exceeds the threshold, a warning is written to the support log and an SNMP trap is thrown. Execution continues and is not affected by the threshold.

This has the same effect as using `SQLSetStmtOption` to set TT_QUERY_THRESHOLD or setting the TimesTen connection attribute `QueryThreshold`. (See “Setting a threshold value for SQL statements” in Oracle TimesTen In-Memory Database C Developer’s Guide.)

```c
int getQueryThreshold()
```

Returns the query threshold value, as described for `setQueryThreshold()`. If no value has been set with `setQueryThreshold()`, this method returns the value of the ODBC connection option TT_QUERY_THRESHOLD (if set) or of the TimesTen connection attribute `QueryThreshold`.

```c
void setMaxRows(const int nMaxRows)
```

This method sets a limit on the number of rows returned by a SELECT statement. If the number of rows in the result set exceeds the set limit, the `TTCmd::FetchNext()` method will return 1 if it fetches beyond the maximum number of rows.

The default is to return all rows. To reset a limit to again return all rows, call `setMaxRows()` with `nMaxRows` set to 0. The limit is only meaningful for SELECT statements.

```c
int getMaxRows()
```

This method returns the current limit of the number of rows returned by a SELECT statement from this `TTCmd` object. A return value of 0 means all rows are returned.
**getRowCount()**

```java
int getRowCount()
```

This method can be called immediately after `Execute()` to return the number of rows that were affected by the executed SQL operation. For example, after execution of a DELETE statement that deletes 10 rows, `getRowCount()` returns 10.

**registerParam()**

```java
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType, int sqltype)
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType, int sqltype, int precision)
void registerParam(int pno, TTCMD_PARAM_INPUTOUTPUT_TYPE inputOutputType, int sqltype, int precision, int scale)
```

Use this method if you need to register a parameter for binding. This is required for OUT and IN OUT parameters and can also be used as appropriate to specify SQL type, precision (maximum number of digits that are used by the data type, where applicable), and scale (maximum number of digits to the right of the decimal point, where applicable). See “Registering parameters” on page 2-9.

**setParam()**

```java
void setParam(int pno, TYPE value)
void setParam(int pno, TYPE* valueP)
void setParam(int pno, TYPE* valueP, int byteLen)
```

All overloaded `setParam()` versions are described in this section.

Each `setParam()` version is used to set the value of a parameter, specified by parameter number, before executing a prepared SQL statement. SQL statements are prepared before use with the `Prepare()` method and are executed with the `Execute()` method. If the SQL statement contains any parameter markers (the '?' character used where a literal constant would be legal), values must be assigned to these parameters before the SQL statement can be executed. The `setParam()` method is used to define a value for each parameter before executing the statement. See “Dynamic parameters” in Oracle TimesTen In-Memory Database SQL Reference.

The first argument passed to `setParam()` is the position of the parameter to be set. The first parameter in a SQL statement is parameter 1. The second argument passed to `setParam()` is the value of the parameter. Overloaded versions of `setParam()` take different data types for the second argument.

The TTClasses library does not support a large set of data type conversions. The appropriate overloaded version of `setParam()` must be called for each parameter in the prepared SQL. Calling the wrong version (attempting to set an integer parameter to a `char*` value, for example) may result in program failure.

Values passed to `setParam()` are copied into internal buffers maintained by the TTCmd object. These buffers are statically allocated and bound by the `Prepare()` method. The parameter value is the value passed into `setParam()` at the time of the `setParam()` call, not the value at the time of a subsequent `Execute()` method call.

*Table 3-2* shows the supported SQL data types and the appropriate versions of `setParam()` to use for each type. Note that SQL data types not mentioned are not supported in this version of TTClasses. For `nchar`, `nvarchar`, and binary types, as shown in the table, the method call includes `byteLen`, an integer value for the number of bytes.
Commonly used TTClasses

See “Binding IN parameters” on page 2-8 and “Binding OUT or IN OUT parameters” on page 2-10 for examples using `setParam()`.

Notes:
- To set the length of the value for a bound parameter, see “setParamLength() (non-batch operations)” on page 3-21.
- To set a NULL value for a bound parameter, see “setParamNull() (non-batch operations)” on page 3-21.

**Table 3–2 setParam() variants for supported data types**

<table>
<thead>
<tr>
<th>Data type</th>
<th>setParam() variants supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT_TINYINT</td>
<td><code>setParam(pno, SQLTINYINT value)</code></td>
</tr>
<tr>
<td>TT_SMALLINT</td>
<td><code>setParam(pno, SQLSMALLINT value)</code></td>
</tr>
<tr>
<td>TT_INTEGER</td>
<td><code>setParam(pno, SQLINTEGER value)</code></td>
</tr>
<tr>
<td>TT_BIGINT</td>
<td><code>setParam(pno, SQLBIGINT value)</code></td>
</tr>
<tr>
<td>BINARY_FLOAT</td>
<td><code>setParam(pno, float value)</code></td>
</tr>
<tr>
<td>REAL</td>
<td><code>setParam(pno, double value)</code></td>
</tr>
<tr>
<td>BINARY_DOUBLE</td>
<td><code>setParam(pno, double value)</code></td>
</tr>
<tr>
<td>NUMBER</td>
<td><code>setParam(pno, char* valueP)</code></td>
</tr>
<tr>
<td>TT_DECIMAL</td>
<td><code>setParam(pno, const char* valueP)</code></td>
</tr>
<tr>
<td></td>
<td><code>setParam(pno, SQLCHAR* value)</code></td>
</tr>
<tr>
<td></td>
<td><code>setParam(pno, SQLTINYINT value)</code></td>
</tr>
<tr>
<td></td>
<td><code>setParam(pno, SQLSMALLINT value)</code></td>
</tr>
<tr>
<td></td>
<td><code>setParam(pno, SQLINTEGER value)</code></td>
</tr>
<tr>
<td></td>
<td><code>setParam(pno, SQLBIGINT value)</code></td>
</tr>
<tr>
<td>Note: The integer type methods are appropriate only for columns declared with the scale parameter set to zero, such as NUMBER(8) or NUMBER(8,0).</td>
<td></td>
</tr>
<tr>
<td>TT_CHAR</td>
<td><code>setParam(pno, char* valueP)</code></td>
</tr>
<tr>
<td>CHAR</td>
<td><code>setParam(pno, const char* valueP)</code></td>
</tr>
<tr>
<td>TT_VARCHAR</td>
<td><code>setParam(pno, SQLCHAR* valueP)</code></td>
</tr>
<tr>
<td>VARCHAR2</td>
<td><code>setParam(pno, SQLCHAR* valueP)</code></td>
</tr>
<tr>
<td>TT_NCHAR</td>
<td><code>setParam(pno, SQLCHAR* valueP, byteLen)</code></td>
</tr>
<tr>
<td>NCHAR</td>
<td><code>setParam(pno, SQLCHAR* valueP)</code></td>
</tr>
<tr>
<td>TT_NVARCHAR</td>
<td><code>setParam(pno, SQLWCHAR* valueP, byteLen)</code></td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td><code>setParam(pno, SQLWCHAR* valueP, byteLen)</code></td>
</tr>
<tr>
<td>BINARY</td>
<td><code>setParam(pno, const void* valueP, byteLen)</code></td>
</tr>
<tr>
<td>VARBINARY</td>
<td><code>setParam(pno, const void* valueP, byteLen)</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>setParam(pno, TIMESTAMP_STRUCT* valueP)</code></td>
</tr>
<tr>
<td>TT_TIMESTAMP</td>
<td><code>setParam(pno, DATE_STRUCT* valueP)</code></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td><code>setParam(pno, DATE_STRUCT* valueP)</code></td>
</tr>
<tr>
<td>TT_DATE</td>
<td><code>setParam(pno, DATE_STRUCT* valueP)</code></td>
</tr>
<tr>
<td>TT_TIME</td>
<td><code>setParam(pno, TIME_STRUCT* valueP)</code></td>
</tr>
</tbody>
</table>
Commonly used TTClasses

Class Descriptions

### setParamLength() (non-batch operations)

```c
void setParamLength(int pno, int byteLen)
```

Sets the length, in bytes, of the bound value for an input parameter specified by parameter number, before execution of the prepared statement.

**Note:** There is also a batch version of this method. See "setParamLength() (batch operations)" on page 3-29.

### setParamNull() (non-batch operations)

```c
void setParamNull(int pno)
```

Sets a value of SQL NULL for a bound input parameter specified by parameter number.

**Note:** There is also a batch version of this method. See "setParamNull() (batch operations)" on page 3-30.

### getParam()

```c
bool getParam(int pno, TYPE* valueP)
bool getParam(int pno, TYPE* valueP, int* byteLenP)
```

Each `getParam()` version is used to retrieve the value of an OUT or IN OUT parameter, specified by parameter number, after executing a prepared SQL statement. SQL statements are prepared before use with the `Prepare()` method and are executed with the `Execute()` method. The `getParam()` method is used to provide a variable of appropriate data type for the value for each output parameter after executing the statement.

The first argument passed to `getParam()` is the position of the parameter for the output value. The first parameter in a SQL statement is parameter 1. The second argument passed to `getParam()` is a variable for the output value. Overloaded versions of `getParam()` take different data types for the second argument.

The `getParam()` method supports the same data types documented for `getColumn()` in Table 3–3. For `nchar`, `nvarchar`, and `binary` types, as shown in that table, the method call includes `byteLenP`, a pointer to an integer value for the number of bytes.

The `getParam()` return value is a boolean that is TRUE if the data is null or FALSE otherwise.

The TTClasses library does not support a large set of data type conversions. The appropriate overloaded version of `getParam()` must be called for each output parameter in the prepared SQL. Calling the wrong version (attempting to use an integer parameter for a `char*` value, for example) may result in program failure.

See "Binding OUT or IN OUT parameters" on page 2-10 for examples using `getParam()`.

For REF CURSORs, the following signature is supported to use a `TTCmd` object as a statement handle for the REF CURSOR (data type SQL_REFCURSOR). See "Working with REF CURSORs" on page 2-13 for information and an example.

```c
bool getParam(int pno, TTCmd** rcCmd)
```
The `getColumn()` method, as well as the `getColumnNullable()` method (described next), can be used to fetch the values for columns of the current row of the result set. Before `getColumn()` or `getColumnNullable()` can be called, the `FetchNext()` method must be called to fetch the next (or first) row from the result set of a `SELECT` statement. SQL statements are executed using the `Execute()` method.

Each `getColumn()` call retrieves the value associated with a particular column. Columns are referred to by ordinal number, with "1" indicating the first column specified in the `SELECT` statement. In all cases the first argument passed to the `getColumn()` method, `cno`, is the ordinal number of the column whose value is to be fetched. The second argument, `valueP`, is a pointer to a variable which is to receive the value of the specified column. The type of this argument varies depending on the type of the column being returned. For `nchar`, `nvarchar`, and binary types, as shown in the table, the method call also includes `byteLenP`, a pointer to an integer value for the number of bytes.

The TTClasses library does not support a large set of data type conversions. The appropriate overloaded version of `getColumn()` must be called for each output column in the prepared SQL. Calling the wrong version (attempting to fetch an integer column into a `char*` value, for example) may result in program failure.

Integer type methods include one of the following functions: `SQLTINYINT`, `SQLSMALLINT`, `SQLINTEGER`, or `SQLBIGINT`. They are appropriate only for columns with the scale parameter set to zero, such as `NUMBER(p)` or `NUMBER(p,0)`. The functions have the following range of precision:

<table>
<thead>
<tr>
<th>Function</th>
<th>Precision Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SQLTINYINT</code></td>
<td>0 &lt;= p &lt;= 2</td>
</tr>
<tr>
<td><code>SQLSMALLINT</code></td>
<td>0 &lt;= p &lt;= 4</td>
</tr>
<tr>
<td><code>SQLINTEGER</code></td>
<td>0 &lt;= p &lt;= 9</td>
</tr>
<tr>
<td><code>SQLBIGINT</code></td>
<td>0 &lt;= p &lt;= 18</td>
</tr>
</tbody>
</table>

To ensure that all values in the column will fit into the variable that the application uses to retrieve information from the database, you can use `SQLBIGINT` for all table columns of data type `NUMBER(p)`, where 0 <= p <= 18. For example:

```c
getColumn(int cno, SQLBIGINT* iP)
```

Table 3–3 shows the supported SQL data types and the appropriate versions of `getColumn()` and `getColumnNullable()` to use for each parameter type.

<table>
<thead>
<tr>
<th>Data type</th>
<th><code>getColumn()</code> variants supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT_TINYINT</td>
<td><code>getColumn(cno, SQLTINYINT* iP)</code></td>
</tr>
<tr>
<td>TT_SMALLINT</td>
<td><code>getColumn(cno, SQLSMALLINT* iP)</code></td>
</tr>
<tr>
<td>TT_INTEGER</td>
<td><code>getColumn(cno, SQLINTEGER* iP)</code></td>
</tr>
<tr>
<td>TT_BIGINT</td>
<td><code>getColumn(cno, SQLBIGINT* iP)</code></td>
</tr>
</tbody>
</table>

Table 3-3 getColumn() variants for supported data types
Other SQL data types are not supported in this release of the TTClasses library.

getColumnNullable()

bool getColumnNullable(int cno, TYPE* valueP)
bool getColumnNullable(int cno, TYPE* valueP, int* byteLenP)

The getColumnNullable() method is similar to the getColumn() method, described previously, and supports the same data types and signatures documented there in Table 3-3. However, in addition to the behavior of getColumn(), the getColumnNullable() method also returns a boolean indicating whether the value is the SQL NULL pseudo-value. If the value is NULL, the second parameter is set to a distinctive value (for example, -9999), and the return value from the method is TRUE. If the value is not NULL, it is returned through the variable pointed to by the second parameter and the getColumnNullable() method returns FALSE.
The `getNextColumn()` method, as well as the `getNextColumnNullable()` method (described next), can be used to fetch the value of the next column of the current row of the result set. Before `getNextColumn()` or `getNextColumnNullable()` can be called, the `FetchNext()` method must be called to fetch the next (or first) row from the result set of a SELECT statement. When you use `getNextColumn()`, the columns are fetched in order. You cannot change the fetch order.

See Table 3–3 for the supported SQL data types and the appropriate method version to use for each data type. This information can be used for `getNextColumn()`, except there is no column number parameter for `getNextColumn()`.  

The `getNextColumnNullable()` method is similar to the `getNextColumn()` method (described previously). However, in addition to the behavior of `getNextColumn()`, the `getNextColumnNullable()` method returns a boolean indicating whether the value is the SQL NULL pseudo-value. If the value is NULL, the second parameter is set to a distinctive value (for example, -9999), and the return value from the method is TRUE. If the value is not NULL, it is returned through the variable pointed to by the second parameter, and the method returns FALSE. When you use `getNextColumnNullable()`, the columns are fetched in order. You cannot change the fetch order.

See Table 3–3 for the supported SQL data types and the appropriate method versions to use for each data type. This information can be used for `getNextColumnNullable()`, except there is no column number parameter for `getNextColumnNullable()`.  

This method provides another way to determine whether the value in column number `cno` of the current row is NULL, returning TRUE if that is the case or FALSE otherwise.  

Also see information about the `getColumnNullable()` method, discussed earlier.  

`getColumnLength()`  

Returns the length, in bytes, of the data in column number `cno` of the current row, not counting the NULL terminator. Or it returns SQL_NULL_DATA if the data is NULL. (For those familiar with ODBC, this is the value stored by ODBC in the last parameter, `pcbValue`, from `SQLBindCol` after a call to `SQLFetch`.) When there is non-NULL data, the length returned is between 0 and the column precision, inclusive. See “`getColumnPrecision()`” on page 3-26.  

For example, assume a VARCHAR2(25) column. If the data is NULL, the length returned is -1. If the data is ‘abcde’, the length returned is 5.
This method is generally useful only when accessing columns of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, BINARY, and VARBINARY.

```
void printColumn(int cno, STDOUT& os, const char* nullString) const
```

This method prints the value in column number `cno` of the current row to the output stream `os`. Use this method for debugging or for demo programs. Use `nullString` to specify what should be printed if the column value is NULL (for example, "NULL" or ":").

**Public methods for obtaining TTCmd object properties**

There are several useful methods for asking questions about properties of the bound input parameters and output columns of a prepared TTCmd object. These methods generally provide meaningful results only when a statement has previously been prepared.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNParameters()</td>
<td>Returns the number of input parameters.</td>
</tr>
<tr>
<td>getParamNullability()</td>
<td>Indicates whether the specified parameter can be NULL.</td>
</tr>
<tr>
<td>getParamType()</td>
<td>Returns the ODBC data type of the specified parameter.</td>
</tr>
<tr>
<td>getParamPrecision()</td>
<td>Returns the precision of the specified parameter in a prepared statement.</td>
</tr>
<tr>
<td>getParamScale()</td>
<td>Returns the scale of the specified parameter in a prepared statement.</td>
</tr>
<tr>
<td>getNColumns()</td>
<td>Returns the number of output columns.</td>
</tr>
<tr>
<td>getColumnNullability()</td>
<td>Indicates whether the specified column can have NULL values.</td>
</tr>
<tr>
<td>getColumnName()</td>
<td>Returns the name of the specified column.</td>
</tr>
<tr>
<td>getColumnType()</td>
<td>Returns the ODBC data type of the specified column.</td>
</tr>
<tr>
<td>getColumnPrecision()</td>
<td>Returns the precision of the specified column.</td>
</tr>
<tr>
<td>getColumnScale()</td>
<td>Returns the scale of the specified column.</td>
</tr>
<tr>
<td>isBeingExecuted</td>
<td>Indicates whether the statement represented by the TTCmd object is being executed.</td>
</tr>
</tbody>
</table>

**getNParameters()**

```
int getNParameters()
```

Returns the number of input parameters for the SQL statement.

**getParamNullability()**

```
int getParamNullability(int pno)
```

Indicates whether parameter number `pno` can have a NULL value. It returns SQL_NO_NULLS, SQL_NULLABLE, or SQLNULLABLE_UNKNOWN.

**Note:** In earlier releases this method returned `bool` instead of `int`. 
getParamType

```c
int getParamType(int pno)
```

Returns the data type of parameter number pno. The value returned is the ODBC type (for example, SQL_INTEGER, SQL_REAL, SQL_BINARY, SQL_CHAR) as found in sql.h. Additional TimesTen types (SQL_WCHAR, SQL_WVARCHAR) can be found in the TimesTen header file timesten.h.

getParamPrecision

```c
int getParamPrecision(int pno)
```

Returns the precision of parameter number pno, referring to the maximum number of digits that are used by the data type. Also see information for getColumnPrecision(), described shortly.

getParamScale

```c
int getParamScale(int pno)
```

Returns the scale of parameter number pno, referring to the maximum number of digits to the right of the decimal point.

getNColumns

```c
int getNColumns()
```

Returns the number of output columns.

columnNullability

```c
int getColumnNullability(int cno)
```

Indicates whether column number cno can have NULL values. It returns SQL_NO_NULLS, SQL_NULLABLE, or SQLNULLABLE_UNKNOWN.

columnName

```c
const char* getColumnName(int cno)
```

Returns the name of column number cno.

columnType

```c
int getColumnType(int cno)
```

Returns the data type of column number cno. The value returned is the ODBC type of the parameter (for example, SQL_INTEGER, SQL_REAL, SQL_BINARY, SQL_CHAR) as found in sql.h. Additional TimesTen types (SQL_WCHAR, SQL_WVARCHAR) can be found in the TimesTen header file timesten.h.

columnPrecision

```c
int getColumnPrecision(int cno)
```

Returns the precision of data in column number cno, referring to the size of the column in the database. For example, for a VARCHAR2(25) column, the precision returned would be 25.
This value is generally interesting only when generating output from table columns of type CHAR, VARCHAR2, NCHAR, NVARCHAR2, BINARY, and VARBINARY.

**getColumnScale()**

```c
int getColumnScale(int cno)
```

Returns the scale of data in column number `cno`, referring to the maximum number of digits to the right of the decimal point.

**isBeingExecuted**

```c
bool isBeingExecuted()
```

Indicates whether the statement represented by the `TTCmd` object is being executed.

**Note:** This method was formerly named `queryBeingExecuted()`. That name is still supported for backward compatibility.

### Public methods for batch operations

TimesTen supports the ODBC function `SQLBindParams` for batch insert, update and delete operations. `TTClasses` provides an interface to the ODBC function `SQLBindParams`.

Performing batch operations with `TTClasses` is similar to performing non-batch operations. SQL statements are first compiled using `PrepareBatch()`. Then each parameter in that statement is bound to an array of values using `BindParameter()`. Finally, the statement is executed using `ExecuteBatch()`.

See the `TTClasses` `bulktest` sample program in the TimesTen Quick Start for an example of using a batch operation. Refer to “About the TimesTen `TTClasses` demos” on page 1-6.

This section describes the `TTCmd` methods that expose the batch INSERT, UPDATE, and DELETE functionality to `TTClasses` users.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PrepareBatch()</code></td>
<td>Prepares batch INSERT, UPDATE, and DELETE statements.</td>
</tr>
<tr>
<td><code>BindParameter()</code></td>
<td>Binds an array of values for one parameter of a statement prepared using <code>PrepareBatch()</code>.</td>
</tr>
<tr>
<td><code>setParamLength()</code></td>
<td>Sets the length, in bytes, of the value of the specified bound parameter before execution of the prepared statement.</td>
</tr>
<tr>
<td><code>setParamNull()</code></td>
<td>Sets the specified bound parameter to NULL before execution of the prepared statement.</td>
</tr>
<tr>
<td><code>ExecuteBatch()</code></td>
<td>Invokes a SQL statement that has been prepared for execution by <code>PrepareBatch()</code>. It returns the number of rows in the batch that were updated.</td>
</tr>
<tr>
<td><code>batchSize()</code></td>
<td>Returns the number of statements in the batch.</td>
</tr>
</tbody>
</table>

**PrepareBatch()**

```c
void PrepareBatch(TTConnection* cP, const char* sqlp, unsigned short batSz)
```
PrepareBatch() is comparable to the Prepare() method but for batch INSERT, UPDATE, or DELETE statements. The TTConnection* and sqlp parameters are used as for Prepare().

The batSz (batch size) parameter specifies the maximum number of insert, update, or delete operations that will be performed using subsequent calls to ExecuteBatch().

A TTStatus object is thrown as an exception if an error occurs.

Note that TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time. See "Considering TimesTen features for access control" on page 2-4.

### BindParameter()

```c
void BindParameter(int pno, unsigned short batSz, TYPE* valueP)
void BindParameter(int pno, unsigned short batSz, TYPE* valueP, size_t maxByteLen)
void BindParameter(int pno, unsigned short batSz, TYPE* valueP, SQLLEN* userByteLenP, size_t maxByteLen)
```

The overloaded BindParameter() method is used to bind an array of values for a specified parameter in a SQL statement compiled using PrepareBatch(). This is to iterate through a batch of repeated executions of the statement with different values. The pno parameter indicates the position in the statement of the parameter to be bound, starting from the left, where the first parameter is 1, the next is 2, and so on.

The batSz (batch size) value of this call must match the batSz value specified in PrepareBatch(), and the bound arrays should contain at least the batSz number of values. You must determine the correct data type for each parameter. Note that if an inappropriate type is specified, a runtime error will be written to the TTClasses global logging facility at the TTLog::TTLOG_ERR logging level.

Table 3-4 shows the supported SQL data types and the appropriate versions of BindParameter() to use for each parameter type.

Before each invocation of ExecuteBatch(), the application should fill the arrays with valid parameter values. Note that you can use the setParamNull() method to set null values, as described in "setParamNull() (batch operations)" on page 3-30.

For the SQL types TT_CHAR, CHAR, TT_VARCHAR, and VARCHAR2, an additional maximum length parameter is required in the BindParameter() call:

- maxByteLen of type size_t is for the maximum length, in bytes, of any value for this parameter position.

For the SQL types TT_NCHAR, NCHAR, TT_NVARCHAR, NVARCHAR2, BINARY and VARBINARY, two additional parameters are required in the BindParameter() call, an array of parameter lengths and a maximum length:

- userByteLenP is an array of SQLLEN parameter lengths, in bytes, to specify the length of each value in the batch for this parameter position in the SQL statement.

Note: To avoid unwanted round trips between client and server when in client/server mode, the PrepareBatch() method performs what is referred to as a "deferred prepare", where the request is not sent to the server until required. See "TimesTen deferred prepare" in Oracle TimesTen In-Memory Database C Developer's Guide for more information.
This array must be at least \( batSz \) in length and filled with valid length values before \( \text{ExecuteBatch()} \) is called. (You can store SQL_NULL_DATA in the array of parameter lengths for a null value, which is equivalent to using the \( \text{setParamNull()} \) batch method.)

- \( \text{maxByteLen} \) is as described above. This indicates the maximum length value that can be specified in any element of the \( \text{userByteLenP} \) array.

For data types where \( \text{userByteLenP} \) is not available (or as an alternative where it is available), you can optionally use the \( \text{setParamLength()} \) batch method to set data lengths, as described in "setParamLength() (batch operations)" on page 3-29, and use the \( \text{setParamNull()} \) batch method to set null values, as described in "setParamNull() (batch operations)" on page 3-30.

See Example 3-5 in "ExecuteBatch()" on page 3-30 for examples of \( \text{BindParameter()} \) use.

### Table 3–4 BindParameter() variants for supported data types

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>BindParameter() variants supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{TT_TINYINT} )</td>
<td>( \text{BindParameter}(pno, batSz, SQLTINYINT* user_tIP) )</td>
</tr>
<tr>
<td>( \text{TT_SMALLINT} )</td>
<td>( \text{BindParameter}(pno, batSz, SQLSMALLINT* user_siP) )</td>
</tr>
<tr>
<td>( \text{TT_INTEGER} )</td>
<td>( \text{BindParameter}(pno, batSz, SQLINTEGER* user_iP) )</td>
</tr>
<tr>
<td>( \text{TT_BIGINT} )</td>
<td>( \text{BindParameter}(pno, batSz, SQLBIGINT* user_biP) )</td>
</tr>
<tr>
<td>( \text{BINARY_FLOAT} )</td>
<td>( \text{BindParameter}(pno, batSz, float* user_fp) )</td>
</tr>
<tr>
<td>( \text{BINARY_DOUBLE} )</td>
<td>( \text{BindParameter}(pno, batSz, double* user_dp) )</td>
</tr>
<tr>
<td>( \text{NUMBER} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{TT_DECIMAL} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{CHAR} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{VARCHAR} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{VARCHAR} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{TT_CHAR} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{NVARCHAR} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{NVARCHAR2} )</td>
<td>( \text{BindParameter}(pno, batSz, char** user_cPP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{BINARY} )</td>
<td>( \text{BindParameter}(pno, batSz, const void** user_binPP, userByteLenP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{VARBINARY} )</td>
<td>( \text{BindParameter}(pno, batSz, const void** user_binPP, userByteLenP, maxByteLen) )</td>
</tr>
<tr>
<td>( \text{DATE} )</td>
<td>( \text{BindParameter}(pno, batSz, DATE_STRUCT* user_dsP) )</td>
</tr>
<tr>
<td>( \text{TIMESTAMP} )</td>
<td>( \text{BindParameter}(pno, batSz, TIMESTAMP_STRUCT* user_tssP) )</td>
</tr>
</tbody>
</table>

#### setParamLength() (batch operations)

\[
\text{void setParamLength}(\text{int pno, unsigned short rowno, int byteLen})
\]

This method sets the length of one of the bound parameter values before a call to \( \text{ExecuteBatch()} \). The \( \text{pno} \) argument specifies the parameter number in the SQL.
Commonly used TTClasses

statement (where the first parameter is number 1). The \texttt{rowno} argument specifies the row number in the array of parameters being bound (where the first row is row number 1). The \texttt{byteLen} parameter specifies the desired length, in bytes, not counting the NULL terminator. Alternatively, \texttt{byteLen} can be set to SQL\_NTS for a null-terminated string. (It can also be set to SQL\_NULL\_DATA, which is equivalent to using the \texttt{setParamNull()} batch method, described next.)

Notes:

- For binary and \texttt{nchar} types, as shown in Table 3–4 above, it is more typical to use the \texttt{BindParameter()} \texttt{userByteLenP} array to set parameter lengths. Be aware that row numbering in the array of parameters being bound starts with 0 in the \texttt{userByteLenP} array but with 1 when you use \texttt{setParamLength()}. There is also a non-batch version of this method. See “setParamLength() (non-batch operations)” on page 3-21.

\texttt{setParamNull()} (batch operations)

\begin{verbatim}
void setParamNull(int pno, unsigned short rowno)
\end{verbatim}

This method sets one of the bound parameter values to NULL before a call to \texttt{ExecuteBatch()}. The \texttt{pno} argument specifies the parameter number in the SQL statement (where the first parameter is number 1). The \texttt{rowno} argument specifies the row number in the array of parameters being bound (where the first row is row number 1).

Notes:

- For binary and \texttt{nchar} types, as shown in Table 3–4 above, there is a \texttt{BindParameter()} \texttt{userByteLenP} array. For these types, you can have a null value by specifying SQL\_NULL\_DATA in this array, which is equivalent to using \texttt{setParamNull()}. Be aware that row numbering in the array of parameters being bound starts with 0 in the \texttt{userByteLenP} array but with 1 when you use \texttt{setParamNull()}. There is also a non-batch version of this method. See “setParamNull() (non-batch operations)” on page 3-21.

\texttt{ExecuteBatch()}

\begin{verbatim}
int ExecuteBatch(unsigned short numRows)
\end{verbatim}

After preparing a SQL statement with \texttt{PrepareBatch()} and calling \texttt{BindParameter()} for each parameter in the SQL statement, use \texttt{ExecuteBatch()} to execute the statement \texttt{numRows} times. The value of \texttt{numRows} must be no more than the \texttt{batSz} (batch size) value specified in the \texttt{PrepareBatch()} and \texttt{BindParameter()} calls, and can be less than \texttt{batSz} as required by application logic.

This method returns the number of rows that were updated, with possible values in the range 0 to \texttt{batSz}, inclusive. (For those familiar with ODBC, this is the third parameter, *\texttt{pirow}, of an ODBC SQLParamOptions call. Refer to ODBC API reference documentation for information about SQLParamOptions.)
Before calling `ExecuteBatch()`, the application should fill the arrays of parameters to be bound by `BindParameter()` with valid values. See “`BindParameter()`” on page 3-28.

A TTStatus object is thrown as an exception if an error occurs (often due to violation of a uniqueness constraint). In this event, the return value is not valid and the batch is incomplete and should generally be rolled back.

Example 3–5 shows how to use the `ExecuteBatch()` method. The bulktest Quick Start demo also shows usage of this method. (See “About the TimesTen TTClasses demos” on page 1-6.)

**Example 3–5 Using the ExecuteBatch() method**

First, create a table with two columns:

```sql
CREATE TABLE batch_table (a TT_INTEGER, b VARCHAR2(100));
```

Here is the sample code. Populate the rows of the table in batches of 50:

```c
#define BATCH_SIZE 50
#define VARCHAR_SIZE 100

int int_array[BATCH_SIZE];
char char_array[BATCH_SIZE][VARCHAR_SIZE];

// Prepare the statement
TTCmd insert;
TTConnection connection;
// (assume a connection has already been established)
try {
    insert.PrepareBatch (&connection, 
                      (const char*)"insert into batch_table values (?,?)", 
                      BATCH_SIZE);
    // Commit the prepared statement
    connection.Commit();
    // Bind the arrays of parameters
    insert.BindParameter(1, BATCH_SIZE, int_array);
    insert.BindParameter(2, BATCH_SIZE, (char **)char_array, VARCHAR_SIZE);
    // Execute 5 batches, inserting 5 * BATCH_SIZE rows into
    // the database
    for (int iter = 0; iter < 5; iter++)
    {
        // Populate the value arrays with values.
        // (A better way of putting meaningful data into
        // the database is to read values from a file,
        // rather than generating them arbitrarily.)
        for (int i = 0; i < BATCH_SIZE; i++)
        {
            int_array[i] = i * iter + i;
            sprintf(char_array[i], "varchar value # %d", i*iter + i);
        }
    }
}
```

Example 3–5: Using the `ExecuteBatch()` method

First, create a table with two columns:

```sql
CREATE TABLE batch_table (a TT_INTEGER, b VARCHAR2(100));
```

Here is the sample code. Populate the rows of the table in batches of 50:

```c
#define BATCH_SIZE 50
#define VARCHAR_SIZE 100

int int_array[BATCH_SIZE];
char char_array[BATCH_SIZE][VARCHAR_SIZE];

// Prepare the statement
TTCmd insert;
TTConnection connection;
// (assume a connection has already been established)
try {
    insert.PrepareBatch (&connection, 
                      (const char*)"insert into batch_table values (?,?)", 
                      BATCH_SIZE);
    // Commit the prepared statement
    connection.Commit();
    // Bind the arrays of parameters
    insert.BindParameter(1, BATCH_SIZE, int_array);
    insert.BindParameter(2, BATCH_SIZE, (char **)char_array, VARCHAR_SIZE);
    // Execute 5 batches, inserting 5 * BATCH_SIZE rows into
    // the database
    for (int iter = 0; iter < 5; iter++)
    {
        // Populate the value arrays with values.
        // (A better way of putting meaningful data into
        // the database is to read values from a file,
        // rather than generating them arbitrarily.)
        for (int i = 0; i < BATCH_SIZE; i++)
        {
            int_array[i] = i * iter + i;
            sprintf(char_array[i], "varchar value # %d", i*iter + i);
        }
    }
}
```
The number of rows updated (\texttt{num_ins} in the example) can be less than \texttt{BATCH_SIZE} if, for example, there is a violation of a uniqueness constraint on one of the columns. You can use code similar to that in Example 3–6 to check for this situation and roll back the transaction as necessary.

Note that TimesTen has features to control database access with object-level resolution for database objects such as tables, views, materialized views, sequences, and synonyms. Access control privileges are checked both when SQL is prepared and when it is executed in the database, with most of the performance cost coming at prepare time. See "Considering TimesTen features for access control" on page 2-4.

**Example 3–6 Using \texttt{ExecuteBatch()} and checking against \texttt{BATCH_SIZE}**

```c++
for (int iter = 0; iter < 5; iter++)
{
    // Populate the value arrays with values.
    // (A better way of putting meaningful data into
    // the database is to read values from a file,
    // rather than generating them arbitrarily.)
    for (int i = 0; i < \texttt{BATCH_SIZE}; i++)
    {
        \texttt{int_array}[i] = i * \texttt{iter} + i;
        sprintf(\texttt{char_array}[i], "varchar value # %d", i*\texttt{iter}+1);
    }
    // now we execute the batch insert statement,
    // which does the work of inserting the entire
    // contents of the integer and char arrays in
    // one operation
    int \texttt{num_ins} = \texttt{insert.ExecuteBatch(\texttt{BATCH_SIZE})};
    \texttt{cerr} << "Inserted " << \texttt{num_ins} << " rows." << \texttt{endl};
    connection.Commit();
}

catch (TTError \texttt{er1})
{
    \texttt{cerr} << \texttt{er1} << \texttt{endl};
}
```

3-32 Oracle TimesTen In-Memory Database TTClasses Guide
System catalog classes

These classes allow you to work with the TimesTen system catalog.

You can use the TTCatalog class to facilitate reading metadata from the system catalog. A TTCatalog object contains easily accessible data structures with the information that was read.

Each TTCatalog object internally contains an array of TTCatalogTable objects. 
Each TTCatalogTable object contains an array of TTCatalogColumn objects and an array of TTCatalogIndex objects.

The following ODBC functions are used inside TTCatalog:

- SQLTables()
- SQLColumns()
- SQLSpecialColumns()
- SQLStatistics()

This section discusses the following classes:

- TTCatalog
- TTCatalogTable
- TTCatalogColumn
- TTCatalogIndex
- TTCatalogSpecialColumn

TTCatalog

The TTCatalog class is the top-level class used for programmatically accessing metadata information about tables in a database. A TTCatalog object contains an internal array of TTCatalogTable objects. Apart from the constructor, all public methods of TTCatalog are used to gain read-only access to that TTCatalogTable array.

The TTCatalog constructor caches the TTConnection* parameter and initializes all the internal data structures appropriately.

TTCatalog (TTConnection* conn)

To use the TTCatalog object, call its fetchCatalogData() method, described shortly. Note that after fetchCatalogData() is called, use of the other TTCatalog methods does not use a database connection.
System catalog classes

Public members
None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetchCatalogData()</td>
<td>Reads the catalogs in the data store for information about tables and</td>
</tr>
<tr>
<td></td>
<td>indexes and stores this information into TTCatalog internal data structures.</td>
</tr>
<tr>
<td>getNumTables()</td>
<td>Returns the total number of tables (user tables plus system tables) in the</td>
</tr>
<tr>
<td></td>
<td>database.</td>
</tr>
<tr>
<td>getNumUserTables()</td>
<td>Returns the number of user tables in the database.</td>
</tr>
<tr>
<td>getNumSysTables()</td>
<td>Returns the number of system tables in the database.</td>
</tr>
<tr>
<td>getTable()</td>
<td>Returns a constant reference to the TTCatalogTable object for the</td>
</tr>
<tr>
<td></td>
<td>specified table.</td>
</tr>
<tr>
<td>getUserTable()</td>
<td>Returns a constant reference to the TTCatalogTable object corresponding to</td>
</tr>
<tr>
<td></td>
<td>the nth user table in the system (where n is specified).</td>
</tr>
<tr>
<td>getTableIndex()</td>
<td>Returns the index in the TTCatalog object for the specified table.</td>
</tr>
</tbody>
</table>

fetchCatalogData()
void fetchCatalogData()

This is the only TTCatalog method that interacts with the data store. It reads the catalogs in the database for information about tables and indexes, storing the information into TTCatalog internal data structures.

Subsequent use of the constructed TTCatalog object is completely offline after it is constructed. It is no longer connected to the database.

You must call this method before you use any of the TTCatalog accessor methods.

This example demonstrates the use of TTCatalog.

Example 3–7 Fetching catalog data

TTConnection conn;
conn.Connect(DSN=TptbmData37);
TTCatalog cat (conn);
cat.fetchCatalogData();

// TTCatalog cat is no longer connected to the database;
// you can now query it through its read-only methods.
cerr << "There are " << cat.getNumTables() << " tables in this database:" << endl;
for (int i=0; i < cat.getNumTables(); i++)
  cerr << cat.getTable(i).getTableOwner() << "." << cat.getTable(i).getTableName() << endl;

ggetNumTables()
int getNumTables()

Returns the total number of tables in the database (user plus system tables).

ggetNumUserTables()

Returns the number of user tables in the database.

ggetNumSysTables()

Returns the number of system tables in the database.

ggetTable()
TTCatalogTable getTable()

Returns a constant reference to the TTCatalogTable object corresponding to the specified table.

ggetUserTable()
TTCatalogTable getUserTable()

Returns a constant reference to the TTCatalogTable object corresponding to the nth user table in the system (where n is specified).

ggetTableIndex()
int getTableIndex()

Returns the index in the TTCatalog object for the specified table.
int getNumUserTables();

Returns the number of user tables in the database.

getNumSysTables()

int getNumSysTables();

Returns the number of system tables in the database.

getTable()

const TTCatalogTable& getTable(const char* owner, const char* tblname)
const TTCatalogTable& getTable(int tno)

Returns a constant reference to the TTCatalogTable object for the specified table. For the first signature, this is for the table named tblname and owned by owner. For the second signature, this is for the table corresponding to table number tno in the system. This is intended to facilitate iteration through all the tables in the system. The order of the tables in this array is arbitrary. Note that the following relationship is asserted to hold:
0 <= tno <= getNumTables()

Also see “TTCatalogTable” on page 3-36.

ggetUserTable();

const TTCatalogTable& getUserTable(int tno)

Returns a constant reference to the TTCatalogTable object corresponding to user table number tno in the system. This method is intended to facilitate iteration through all of the user tables in the system. The order of the user tables in this array is arbitrary. Note that the following relationship is asserted to hold:
0 <= tno <= getNumUserTables();

getTableIndex()

int getTableIndex(const char* owner, const char* tblname) const

This method fetches the index in the TTCatalog object for the specified owner.tblname object. It returns -2 if owner.tblname does not exist. It returns -1 if fetchCatalogData() was not called first.

Example 3–8 retrieves information about the TTUSER.MYDATA table from a TTCatalog object. You can then call methods of TTCatalogTable, described next, to get information about this table.

Example 3–8 Retrieving table information from a catalog

TTConnection conn;
conn.Connect(...);
TTCatalog cat(&conn);
cat.fetchCatalogData();

Note: There is no equivalent method for system tables.
int idx = cat.getTableIndex("TTUSER", "MYDATA");
if (idx < 0) {
    cerr << "Table TTUSER.MYDATA does not exist.\n" << endl;
    return;
}
TTCatalogTable table = cat.getTable(idx);

**TTCatalogTable**

A TTCatalogTable object is retrieved through the TTCatalog::getTable() method and stores all metadata information about the columns and indexes of a table.

**Public members**

None.

**Public methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTableOwner()</td>
<td>Returns the owner of the table.</td>
</tr>
<tr>
<td>getTableName()</td>
<td>Returns the name of the table.</td>
</tr>
<tr>
<td>getNumColumns()</td>
<td>Returns the number of columns in the table.</td>
</tr>
<tr>
<td>getNumIndexes()</td>
<td>Returns the number of indexes on the table.</td>
</tr>
<tr>
<td>getColumn()</td>
<td>Returns a constant reference to the TTCatalogColumn corresponding to the ith column in the table.</td>
</tr>
<tr>
<td>getIndex()</td>
<td>Returns a constant reference to the TTCatalogIndex object corresponding to the nth index in the table, where n is specified.</td>
</tr>
<tr>
<td>getType()</td>
<td>Returns the table type as from an ODBC SQLTables call.</td>
</tr>
<tr>
<td>isSystemTable()</td>
<td>Returns TRUE if the table is a system table.</td>
</tr>
<tr>
<td>isUserTable()</td>
<td>Returns TRUE if the table is a user table.</td>
</tr>
<tr>
<td>getNumSpecialColumns()</td>
<td>Returns the number of special columns in this table. See &quot;TTCatalogSpecialColumn&quot; on page 3-40.</td>
</tr>
<tr>
<td>getSpecialColumn()</td>
<td>Returns a special column (TTCatalogSpecialColumn object) from this table, according to the specified column number.</td>
</tr>
</tbody>
</table>

**getTableOwner()**

*const char* getTableOwner()

Returns the owner of the table.

**getTableName()**

*const char* getTableName()

Returns the name of the table.

**getNumColumns()**

*int* getNumColumns()
Returns the number of columns in the table.

得到表数

```cpp
int getNumColumns();
```

Returns the number of indexes on the table.

得到索引数

```cpp
int getNumIndexes();
```

Returns a constant reference to the `TTCatalogColumn` object corresponding to column number `cno` in the table. This method is intended to facilitate iteration through all the columns in the table.

得到指定列的引用

```cpp
const TTCatalogColumn& getColumn(int cno);
```

Note that the following relationship is asserted to hold:

```cpp
0 <= cno <= getNumColumns();
```

Returns a constant reference to the `TTCatalogIndex` corresponding to index number `num` in the table. This method is intended to facilitate iteration through all the indexes of the table. The order of the indexes of a table in this array is arbitrary.

得到索引的引用

```cpp
const TTCatalogIndex& getIndex(int num);
```

Note that the following relationship is asserted to hold:

```cpp
0 <= num <= getNumIndexes();
```

Returns the table type of this `TTCatalogTable` object, as from an ODBC SQLTables call. For TimesTen this may be TABLE, SYSTEM TABLE, VIEW, or SYNONYM.

得到表的类型

```cpp
const char* getTableType() const;
```

Returns TRUE if the table is a system table (owned by SYS or TTREP). It returns FALSE otherwise.

是否是系统表

```cpp
bool isSystemTable();
```

The `isSystemTable()` method and `isUserTable()` method (described next) are useful for applications that iterate over all tables in a database after a call to `TTCatalog::fetchCatalogData()`, so that you can filter or annotate tables to differentiate the system and user tables. The TTClasses demo program catalog provides an example of how this can be done. (See “About the TimesTen TTClasses demos” on page 1-6.)

是否是用户表

```cpp
bool isUserTable();
```

Returns TRUE if this is a user table, which is to say it is not a system table. It returns FALSE otherwise. Note that `isUserTable()` returns the opposite of `isSystemTable()` for any table. The description of `isSystemTable()`, immediately preceding, discusses the usage and usefulness of these methods.
getNumSpecialColumns()

int getNumSpecialColumns()

Returns the number of special columns in this TTCatalogTable object. Because TimesTen supports only rowid special columns, this always returns 1. Also see "TTCatalogSpecialColumn" on page 3-40.

getSpecialColumn()

const TTCatalogSpecialColumn& getSpecialColumn(int num) const

Returns a special column (TTCatalogSpecialColumn object) from this TTCatalogTable object, according to the specified column number. In TimesTen this can only be a rowid pseudocolumn.

TTCatalogColumn

The TTCatalogColumn class is used to store all metadata information about a single column of the TTCatalogTable object it is associated with.

Public members

None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnName()</td>
<td>Return the name of the column.</td>
</tr>
<tr>
<td>getDataType()</td>
<td>Returns an integer representing the ODBC SQL data type of the column.</td>
</tr>
<tr>
<td>getTypeName()</td>
<td>Returns the database-dependent name of the type returned by getDataType().</td>
</tr>
<tr>
<td>getNullable()</td>
<td>Indicates whether the column can have NULL values.</td>
</tr>
<tr>
<td>getLength()</td>
<td>Returns the length of the column, in bytes.</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>Returns the precision of the column.</td>
</tr>
<tr>
<td>getScale()</td>
<td>Returns the scale of the column.</td>
</tr>
<tr>
<td>getRadix()</td>
<td>Returns the radix of the column.</td>
</tr>
</tbody>
</table>

columnName()

const char* columnName()

Returns the name of the column.

dataType()

int dataType()

Returns an integer representing the data type of the column. This is the standard ODBC SQL Type.

typeName()


const char* getTypeName()

Returns the database-dependent name of the type returned by getDataType().

getNullable()

int getNullable()

Indicates whether the column can have NULL values. It returns SQL_NO_NULLS, SQL_NULLABLE, or SQL_NULLABLE_UNKNOWN.

getLength()

int getLength()

Returns the length of data in the column, in bytes.

getPrecision()

int getPrecision()

Returns the precision of data in the column, referring to the maximum number of digits that are used by the data type.

getScale()

int getScale()

Returns the scale of data in the column, referring to the maximum number of digits to the right of the decimal point.

getRadix()

int getRadix()

Returns the radix of the column, according to ODBC SQLColumns functionality.

TTCatalogIndex

Used to store all information about an index of the TTCatalogTable object it is associated with.

Public members

None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getIndexName()</td>
<td>Returns the name of the index.</td>
</tr>
<tr>
<td>getIndexOwner()</td>
<td>Returns the owner of the index.</td>
</tr>
<tr>
<td>getTableName()</td>
<td>Returns the name of the table for which the index was created.</td>
</tr>
<tr>
<td>getType()</td>
<td>Returns the type of the index.</td>
</tr>
<tr>
<td>isUnique()</td>
<td>Indicates whether the index is a unique index.</td>
</tr>
<tr>
<td>getNumColumns()</td>
<td>Returns the number of columns in the index.</td>
</tr>
</tbody>
</table>
System catalog classes

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getIndexName()</td>
<td>Returns the name of the index.</td>
</tr>
<tr>
<td>getIndexOwner()</td>
<td>Returns the owner of the index.</td>
</tr>
<tr>
<td>getTableName()</td>
<td>Returns the name of the table for which the index was created.</td>
</tr>
<tr>
<td>getType()</td>
<td>Returns the type of the index. For TimesTen, the allowable values are PRIMARY_KEY, HASH_INDEX (the same as PRIMARY_KEY), and TTREE_INDEX.</td>
</tr>
<tr>
<td>isUnique()</td>
<td>Returns TRUE if the index is a unique index, or FALSE otherwise.</td>
</tr>
<tr>
<td>getNumColumns()</td>
<td>Returns the number of columns in the index.</td>
</tr>
<tr>
<td>getColumnName()</td>
<td>Returns the name of column number num in the index.</td>
</tr>
<tr>
<td>getCollation()</td>
<td>Returns the collation of column number num in the index. Values returned are &quot;A&quot; for ascending and &quot;D&quot; for descending index order.</td>
</tr>
</tbody>
</table>

**TTCatalogSpecialColumn**

This class is a wrapper for results from an ODBC SQLSpecialColumns call. In TimesTen, a rowid pseudocolumn is the only type of special column supported, so a
TTCatalogSpecialColumn object can only contain information about rowids of a TTCatalogTable object.

**Usage**

Obtain a TTCatalogSpecialColumn object by calling the getSpecialColumn() method on the relevant TTCatalogTable object.

**Public members**

None.

**Public methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColumnname()</td>
<td>Returns the name of this special column.</td>
</tr>
<tr>
<td>getDataType()</td>
<td>Returns the data type of this special column, as an integer.</td>
</tr>
<tr>
<td>getLength()</td>
<td>Returns the length of data in this special column, in bytes.</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>Returns the precision of the column.</td>
</tr>
<tr>
<td>getScale()</td>
<td>Returns the scale of the column.</td>
</tr>
<tr>
<td>getTypeName()</td>
<td>Returns the data type of the column, as a character string.</td>
</tr>
</tbody>
</table>

**getColumnname()**

const char* getColumnName()

Returns the name of this special column.

**getDataType()**

int getDataType()

Returns an integer representing the ODBC SQL data type of this special column. In TimesTen this can only be SQL_ROWID.

**getLength()**

int getLength()

Returns the length of data in this special column, in bytes.

**getPrecision()**

int getPrecision()

Returns the precision for data in this special column, referring to the maximum number of digits that are used by the data type.

**getScale()**

int getScale()

Returns the scale for data in this special column, referring to the maximum number of digits to the right of the decimal point.

**getTypeName()**
const char* getTypeName()

Returns the data type name that corresponds to the ODBC SQL data type value returned by getDataType(). In TimesTen this can only be ROWID.

XLA classes

TTClasses provides a set of classes that make it easy to write applications that use the TimesTen Transaction Log API (XLA).

XLA is a set of C callable functions that allow an application to monitor changes made to one or more tables in a TimesTen data store. Whenever another application changes a monitored table, the application using XLA is informed of the changes. For more information about XLA, see "XLA and TimesTen Event Management" in Oracle TimesTen In-Memory Database C Developer's Guide.

The XLA classes supports as many XLA columns as the maximum number of columns supported by TimesTen. For more information, see "System Limits" in Oracle TimesTen In-Memory Database Reference.

Important: As noted in "Considerations when using an ODBC driver manager" on page 1-5, XLA functionality does not work in TTClasses when you use an ODBC driver manager.

This section discusses the following classes:

- TTXlaPersistConnection
- TTXlaRowViewer
- TTXlaTableHandler
- TTXlaTableList
- TTXlaTable
- TTXlaColumn

TTXlaPersistConnection

Use TTXlaPersistConnection to create an XLA connection to a TimesTen data store.

Usage

An XLA application can create multiple TTXlaPersistConnection objects if needed. Each TTXlaPersistConnection object must be associated with its own bookmark, which is specified at connect time and must be maintained through the ackUpdates() and deleteBookmarkAndDisconnect() methods. Most applications require only one or at most two XLA bookmarks.

After an XLA connection is established, the application should enter a loop in which the fetchUpdatesWait() method is called repeatedly until application termination. This loop should fetch updates from XLA as rapidly as possible to ensure that the transaction log does not fill up available disk space.
After processing a batch of updates, the application should call `ackUpdates()` in order to acknowledge those updates and get ready for the next call to `fetchUpdatesWait()`. A batch of updates can be replayed using the `setBookmarkIndex()` and `getBookmarkIndex()` methods. Also, if the XLA application disconnects after `fetchUpdatesWait()` but before `ackUpdates()`, the next connection (with the same bookmark name) that calls `fetchUpdatesWait()` will see that same batch of updates.

Updates that occur while a `TTXlaPersistConnection` object is disconnected from the data store are not lost. They are stored in the transaction log until another `TTXlaPersistConnection` object connects with the same bookmark name.

Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See "Access control for connections" on page 2-6. In addition, the XLA privilege is required to create an XLA connection.

Public members
None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect()</td>
<td>Connects with the specified bookmark, or creates one if it does not exist (depending on the method signature).</td>
</tr>
<tr>
<td>deleteBookmarkAndDisconnect()</td>
<td>Deletes the bookmark and disconnects from the data store.</td>
</tr>
<tr>
<td>Disconnect()</td>
<td>Closes an XLA connection to a TimesTen data store.</td>
</tr>
<tr>
<td>ackUpdates()</td>
<td>Advances the bookmark to the next set of updates.</td>
</tr>
<tr>
<td>getBookmarkIndex()</td>
<td>Stores the current position in the transaction log.</td>
</tr>
<tr>
<td>setBookmarkIndex()</td>
<td>Returns to the saved transaction log index.</td>
</tr>
<tr>
<td>fetchUpdatesWait()</td>
<td>Fetches updates to the transaction log within the specified wait period.</td>
</tr>
</tbody>
</table>

Notes:
- The transaction log is in a file system location according to the `LogDir` connection attribute setting, if specified, or the `Datastore` attribute setting. Refer to "Data Store Attributes" in Oracle TimesTen In-Memory Database Reference.
- Each bookmark establishes its own log hold on the transaction log. (See "ttLogHolds" in Oracle TimesTen In-Memory Database Reference for related information.) If any bookmark is not moved forward periodically, transaction logs cannot be purged by checkpoint operations. This can fill up disk space over time.

Notes:
- The transaction log is in a file system location according to the `LogDir` connection attribute setting, if specified, or the `Datastore` attribute setting. Refer to "Data Store Attributes" in Oracle TimesTen In-Memory Database Reference.
- Each bookmark establishes its own log hold on the transaction log. (See "ttLogHolds" in Oracle TimesTen In-Memory Database Reference for related information.) If any bookmark is not moved forward periodically, transaction logs cannot be purged by checkpoint operations. This can fill up disk space over time.
Each XLA connection has a bookmark name associated with it, so that after disconnecting and reconnecting, the same place in the transaction log can be found. The name for the bookmark of a connection is specified in the bookmarkStr parameter.

For the first set of methods documented above, the createBookmarkFlag boolean parameter indicates whether the specified bookmark is new or was previously created. If you specify that a bookmark is new (createBookmarkFlag==true) and it already exists, an error will be returned. Similarly, if you specify that a bookmark already exists (createBookmarkFlag==false) and it does not already exist, an error will be returned.

For the second set of methods, without the boolean parameter, TTClasses first tries to connect reusing the supplied bookmark (behavior equivalent to createBookmarkFlag==false). If that bookmark does not exist, TTClasses then tries to connect and create a new bookmark with the name bookmarkStr (behavior equivalent to createBookmarkFlag==true). These methods are provided as a convenience, to simplify XLA connection logic if you would rather not concern yourself with whether the XLA bookmark exists.

In either mode (with or without the boolean parameter), specify a user name and password, either through the connection string or the separate parameters, or specify a DRIVER_COMPLETION_ENUM value. Refer to “TTConnection” on page 3-6 for information about DRIVER_COMPLETION_ENUM.

Note that privilege to connect to a TimesTen data store must be granted to users through the CREATE SESSION privilege, either directly or through the PUBLIC role. See “Access control for connections” on page 2-6. In addition, the XLA privilege is required to create an XLA connection.

**Note:** Only one XLA connection can connect with a given bookmark name. An error will be returned if multiple connections try to connect to the same bookmark.

deleteBookmarkAndDisconnect()

This method first deletes the bookmark that is currently associated with the connection, so that the data store no longer keeps records relevant to that bookmark, then disconnects from the data store.

Disconnect()

virtual void Disconnect()
This method closes an XLA connection to a TimesTen data store. The XLA bookmark persists after you call this method. If you want to delete the bookmark and disconnect from the data store, use `deleteBookmarkAndDisconnect()` instead.

ackUpdates()

void ackUpdates();

Use this method to advance the bookmark to the next set of updates. After you have acknowledged a set of updates, the updates cannot be viewed again by this bookmark.

See the descriptions of `getBookmarkIndex()` and `setBookmarkIndex()`, following shortly, for information about replaying a set of updates.

Applications should acknowledge updates when a batch of XLA records have been read and processed, so that the transaction log does not fill up available disk space; however, do not call `ackUpdates()` too frequently, because it is a relatively expensive operation.

If an application uses XLA to read a batch of records and then a failure occurs prior to `ackUpdates()` being called, the records will be retrieved when the application reestablishes an XLA connection.

getBookmarkIndex()

void getBookmarkIndex();

This method acquires the current bookmark location, storing the current position in the transaction log.

setBookmarkIndex()

void setBookmarkIndex();

This method returns to the saved transaction log index, restoring the bookmark to the previously acquired bookmark location. Use this method to replay a batch of records multiple times.

Note that `ackUpdates()` invalidates the stored transaction log placeholder. After `ackUpdates()`, a call to `setBookmarkIndex()` returns an error because it is no longer possible to go back to the previously acquired bookmark location.

fetchUpdatesWait()

void fetchUpdatesWait(ttXlaUpdateDesc_t*** arry, int maxrecs, int* recsP, int seconds);

This method is used by an XLA application to fetch a set of records describing changes to a data store. A list of `ttXlaUpdateDesc_t` structures is returned. If there are no XLA updates to be fetched, this method waits the specified number of seconds before returning.

Specify the number of seconds to wait, `seconds`, and the maximum number of records to receive, `maxrecs`. The method returns the number of records actually
XLA classes

received, \texttt{recsP}, and an array of pointers that point to structures defining the changes, \texttt{ary}.

The \texttt{ttXlaUpdateDesc_t} structures that are returned by this method are defined in the XLA specification. No C++ object-oriented encapsulation of these methods is provided. Typically, after calling \texttt{fetchUpdatesWait()}, an application processes these \texttt{ttXlaUpdateDesc_t} structures in a sequence of calls to \texttt{TTXlaTableList::HandleChange()}. See “\texttt{ttXlaUpdateDesc_t}” in Oracle TimesTen In-Memory Database C Developer’s Guide for more information about that data structure.

\textbf{TTXlaRowViewer}

The \texttt{TTXlaRowViewer} class allows application developers to examine XLA change notification record structures and old and new column values.

\textbf{Usage}

Methods of this class are used to examine column values from row images contained in change notification records. Also see related information about the \texttt{TTXlaTable} class (“\texttt{TTXlaTable}” on page 3-53).

Before a row can be examined, the \texttt{TTXlaRowViewer} object must be associated with a row using the \texttt{setTuple()} method, which is invoked inside the \texttt{TTXlaTableHandler::HandleInsert()}, \texttt{HandleUpdate()}, or \texttt{HandleDelete()} method, or by a user-written overloaded method. Columns can be checked for null values using the \texttt{isNull()} method. Non-NULL column values can be examined using one of the overloaded \texttt{Get()} methods.

\textbf{Public members}

None.

\textbf{Public methods}

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{setTuple()}</td>
<td>Associates the \texttt{TTXlaRowViewer} object with the specified row image.</td>
</tr>
<tr>
<td>\texttt{isNull()}</td>
<td>Indicates whether the specified column in a row image is NULL.</td>
</tr>
<tr>
<td>\texttt{Get()}</td>
<td>Fetches the value of the specified column in a row image.</td>
</tr>
<tr>
<td>\texttt{getColumn()}</td>
<td>Returns the specified column from the \texttt{TTXlaRowViewer} object.</td>
</tr>
<tr>
<td>\texttt{columnPrec()}</td>
<td>Returns the precision of the specified column.</td>
</tr>
<tr>
<td>\texttt{columnScale()}</td>
<td>Returns the scale of the specified column.</td>
</tr>
<tr>
<td>\texttt{isColumnTTTimestamp()}</td>
<td>Indicates whether the specified column is a TT_TIMESTAMP column.</td>
</tr>
<tr>
<td>\texttt{numUpdatedCols()}</td>
<td>Returns the number of columns in the \texttt{TTXlaRowViewer} object that have been updated.</td>
</tr>
<tr>
<td>\texttt{updatedCol()}</td>
<td>Returns the column number of a column that has been updated, typically during iteration through all updated columns.</td>
</tr>
</tbody>
</table>

\begin{verbatim}
void setTuple(ttXlaUpdateDesc_t* updateDescP, int whichTuple)
\end{verbatim}
This method associates the TTXlaRowViewer object with a particular row image. You would typically call it only when overloading the TTXlaTableHandler::HandleChange() method. The Quick Start xlasubscriber1 demo provides an example of its usage. (See “About the TimesTen TTC classes demos” on page 1-6.)

The ttXlaUpdateDesc_t structures that are returned by TTXlaPersistConnection::fetchUpdatesWait() contain either zero, one, or two rows. Note the following:

- Structures that define a row that was inserted into a table contain the row image of the inserted row.
- Structures that define a row that was deleted from a table contain the row image of the deleted row.
- Structures that define a row that was updated in a table contain the images of the row before and after the update.
- Structures that define other changes to the table or the data store contain no row images. For example, structures reporting that an index was dropped contain no row images.

The setTuple() method takes two arguments:

- A pointer to a particular ttXlaUpdateDesc_t structure defining a database change.
- An integer specifying which type of row image in the update structure should be examined. The following are valid values:
  - INSERTED_TUP: Examine the inserted row.
  - DELETED_TUP: Examine the deleted row.
  - UPDATE_OLD_TUP: Examine the row before it was updated.
  - UPDATE_NEW_TUP: Examine the row after it was updated.

isNull()

bool isNull(int cno)

Indicates whether a particular column in a row image is NULL, returning TRUE if so or FALSE if not. The cno parameter is the column number for the column to be examined.

Get()

void Get(int cno, TYPE* valueP)
void Get(int cno, TYPE* valueP, int* byteLenP)

Fetches the value of a particular column in a row image. These methods are very similar to the TTCmd::getColumn() methods.

The cno parameter specifies the column number.

Table 3–5 shows the supported SQL data types and the appropriate versions of Get() to use for each data type. Design the application according to the kind of data that is stored. For example, data of type NUMBER(9,0) can be accessed by the Get(int, int*) method without loss of data.
XLA classes

### Table 3–5  Get() variants for supported data types

<table>
<thead>
<tr>
<th>XLA data type</th>
<th>Database data type</th>
<th>Get variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTXLA_CHAR_TT</td>
<td>TT_CHAR</td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td>TTXLA_NCHAR_TT</td>
<td>TT_NCHAR</td>
<td>Get(cno, SQLCHAR** wcPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_VARCHAR_TT</td>
<td>TT_VARCHAR</td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td>TTXLA_NVARCHAR_TT</td>
<td>TT_NVARCHAR</td>
<td>Get(cno, SQLCHAR** wcPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_TINYINT</td>
<td>TT_TINYINT</td>
<td>Get(cno, SQLTINYINT* iP)</td>
</tr>
<tr>
<td>TTXLA_SMALLINT</td>
<td>TT_SMALLINT</td>
<td>Get(cno, short* iP)</td>
</tr>
<tr>
<td>TTXLA_INTEGER</td>
<td>TT_INTEGER</td>
<td>Get(cno, int* iP)</td>
</tr>
<tr>
<td>TTXLA_BIGINT</td>
<td>TT_BIGINT</td>
<td>Get(cno, SQLBIGINT* biP)</td>
</tr>
<tr>
<td>TTXLA_BINARY_FLOAT</td>
<td>BINARY_FLOAT</td>
<td>Get(cno, float* iP)</td>
</tr>
<tr>
<td>TTXLA_BINARY_DOUBLE</td>
<td>BINARY_DOUBLE</td>
<td>Get(cno, double* iP)</td>
</tr>
<tr>
<td>TTXLA_DECIMAL_TT</td>
<td>TT_DECIMAL</td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td>TTXLA_TIME</td>
<td>TT_TIME</td>
<td>Get(cno, TIME_STRUCT* iP)</td>
</tr>
<tr>
<td>TTXLA_DATE_TT</td>
<td>TT_DATE</td>
<td>Get(cno, DATE_STRUCT* iP)</td>
</tr>
<tr>
<td>TTXLA_TIMESTAMP_TT</td>
<td>TT_TIMESTAMP</td>
<td>Get(cno, TIMESTAMP_STRUCT* tsP)</td>
</tr>
<tr>
<td>TTXLA_BINARY</td>
<td>BINARY</td>
<td>Get(cno, const void** binPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_VARBINARY</td>
<td>VARBINARY</td>
<td>Get(cno, const void** binPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_NUMBER</td>
<td>NUMBER</td>
<td>Get(cno, double* iP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get(cno, short* iP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get(cno, int* iP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get(cno, SQLNVARCHAR* biP)</td>
</tr>
<tr>
<td>TTXLA_DATE</td>
<td>DATE</td>
<td>Get(cno, TIMESTAMP_STRUCT* tsP)</td>
</tr>
<tr>
<td>TTXLA_TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Get(cno, TIMESTAMP_STRUCT* tsP)</td>
</tr>
<tr>
<td>TTXLA_CHAR</td>
<td>CHAR</td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td>TTXLA_NCHAR</td>
<td>NCHAR</td>
<td>Get(cno, SQLCHAR** wcPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_VARCHAR</td>
<td>VARCHAR</td>
<td>Get(cno, char** cPP)</td>
</tr>
<tr>
<td>TTXLA_NVARCHAR</td>
<td>NVARCHAR2</td>
<td>Get(cno, SQLCHAR** wcPP, byteLenP)</td>
</tr>
<tr>
<td>TTXLA_FLOAT</td>
<td>FLOAT</td>
<td>Get(cno, double* iP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get(cno, char** cPP)</td>
</tr>
</tbody>
</table>

**getColumn()**

```cpp
const TTXlaColumn* getColumn(u_int cno) const
```

Returns a TTXlaColumn object with metadata for column number cno in the TTXlaRowViewer object. (See "TTXlaColumn" on page 3-54.)

**columnPrec()**

```cpp
int columnPrec(int cno)
```

---

3-48  Oracle TimesTen In-Memory Database TTClasses Guide
XLA classes

Returns the precision of data in column number \texttt{cno}, referring to the maximum number of digits that are used by the data type.

\texttt{columnScale()}

```c
int columnScale(int cno)
```

Returns the scale of data in column number \texttt{cno}, referring to the maximum number of digits to the right of the decimal point.

\texttt{isColumnTTTimestamp()}

```c
bool isColumnTTTimestamp(int cno)
```

Returns \texttt{TRUE} if column number \texttt{cno} is a \texttt{TT_TIMESTAMP} column, or \texttt{FALSE} otherwise.

\texttt{numUpdatedCols()}

```c
SQLUSMALLINT numUpdatedCols()
```

Returns the number of columns that have been updated in the \texttt{TTXlaRowViewer} object.

\texttt{updatedCol()}

```c
SQLUSMALLINT updatedCol(u_int cno)
```

Returns the column number of a column that has been updated. For the input parameter you can iterate from 1 through \texttt{N} where \texttt{N} is the number returned by \texttt{numUpdatedCols()}. Example 3–9 shows a snippet from the TimesTen Quick Start demo \texttt{xlasubscriber1}, where \texttt{updatedCol()} is used with \texttt{numUpdatedCols()} to retrieve each column that has been updated. (See "About the TimesTen TTClasses demos" on page 1-6.)

Example 3–9 Using \texttt{TTXlaRowViewer::numUpdatedCols()} and \texttt{updatedCol()}

```c
void SampleHandler::HandleUpdate(ttXlaUpdateDesc_t* )
{
  cerr << row2.numUpdatedCols() << " column(s) updated: ";
  for ( int i = 1; i <= row2.numUpdatedCols(); i++ )
  {
    cerr << row2.updatedCol(i) << ";";
    if ( i != 1 )
      cerr << row2.getColumn(row2.updatedCol(i)-1)->getColName() << ",";
  }
  cerr << endl;
}
```

\texttt{TTXlaTableHandler}

The \texttt{TTXlaTableHandler} class provides methods that enable and disable change tracking for a table. Methods are also provided to handle update notification records from XLA. It is intended as a base class from which application developers write customized classes to process changes to a particular table.

The constructor associates the \texttt{TTXlaTableHandler} object with a particular table and initializes the \texttt{TTXlaTable} data member contained within the \texttt{TTXlaTableHandler} object.
XLA classes

TTXlaTableHandler(TTXlaPersistConnection& conn, const char* ownerP,
     const char* nameP)

Also see “TTXlaTable” on page 3-53.

Usage
Application developers can derive one or more classes from TTXlaTableHandler and can put most of the application’s logic in the HandleInsert(), HandleDelete(), and HandleUpdate() methods of that class.

One strategy is to derive multiple classes from TTXlaTableHandler, one for each table. Business logic to handle changes to customer data might be implemented in a CustomerTableHandler class, for example, while business logic to handle changes to order data might be implemented in an OrderTableHandler class.

Another strategy is to derive one or more generic classes from TTXlaTableHandler to handle various scenarios. For example, a generic class derived from TTXlaTableHandler could be used to publish changes using a publish/subscribe system.

See the xlasubscriber1 and xlasubscriber2 demos in the TimesTen Quick Start for examples of classes that extend TTXlaTableHandler. (Refer to “About the TimesTen TTClasses demos” on page 1-6.)

Public members
None

Protected members

<table>
<thead>
<tr>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTXlaTable tbl</td>
<td>The metadata associated with the table being handled.</td>
</tr>
<tr>
<td>TTXlaRowViewer row</td>
<td>Used to view the row being inserted or deleted, or the old image of the row being updated, in user-written HandleInsert(), HandleDelete(), and HandleUpdate() methods.</td>
</tr>
<tr>
<td>TTXlaRowViewer row2</td>
<td>Used to view the new image of the row being updated in user-written HandleUpdate() methods.</td>
</tr>
</tbody>
</table>

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnableTracking()</td>
<td>Enables XLA update tracking for the underlying table.</td>
</tr>
<tr>
<td>DisableTracking()</td>
<td>Disables XLA update tracking for the underlying table.</td>
</tr>
<tr>
<td>HandleChange()</td>
<td>Dispatches a record from ttxlaUpdateDesc_t to the appropriate handling routine for processing.</td>
</tr>
<tr>
<td>HandleDelete()</td>
<td>Invoked when the HandleChange() method is called to process a delete operation.</td>
</tr>
<tr>
<td>HandleInsert()</td>
<td>Invoked when the HandleChange() method is called to process an insert operation.</td>
</tr>
<tr>
<td>HandleUpdate()</td>
<td>Invoked when the HandleChange() method is called to process an update operation.</td>
</tr>
</tbody>
</table>
EnableTracking()

virtual void EnableTracking()

Enables XLA update tracking for the underlying table. Until this method is called, XLA will not return information about changes to the table.

DisableTracking()

virtual void DisableTracking()

Disables XLA update tracking for the underlying table. After this method is called, XLA will not return information about changes to the table.

HandleChange()

virtual void HandleChange(ttXlaUpdateDesc_t* updateDescP)
virtual void HandleChange(ttXlaUpdateDesc_t* updateDescP, void* pData)

Dispatches a ttXlaUpdateDesc_t object to the appropriate handling routine for processing. The update description is analyzed to determine if it is for a delete, insert or update operation. The appropriate handing method is then called—HandleDelete(), HandleInsert(), or HandleUpdate().

Classes that inherit from TTXlaTableHandler may wish to use the optional pData parameter when they overload the TTXlaTableHandler::HandleChange() method. This optional parameter is useful for determining whether the batch of XLA records that was just processed ends on a transaction boundary. Knowing this will help an application decide the appropriate time to invoke TTConnection::ackUpdates(). See “Acknowledging XLA updates at transaction boundaries” on page 2-18 for an example that uses the pData parameter.

Also see “HandleChange()” on page 3-53 for TTXlaTableList objects.

HandleDelete()

virtual void HandleDelete(ttXlaUpdateDesc_t* updateDescP) = 0

This method will be invoked whenever the HandleChange() method is called to process a delete operation.

This method is not implemented in the TTXlaTableHandler base class. It must be provided by any classes derived from it, with appropriate logic to handle deleted rows.

The row that was deleted from the table is available through the protected member row (of type TTXlaRowViewer).

HandleInsert()

virtual void HandleInsert(ttXlaUpdateDesc_t* updateDescP) = 0

This method will be invoked whenever the HandleChange() method is called to process a insert operation.

generateSQL() Returns the SQL associated with a given XLA record.
This method is not implemented in the TTXlaTableHandler base class. It must be provided by any classes derived from it, with appropriate logic to handle inserted rows.

The row that was inserted into the table is available through the protected member row (of type TTXlaRowViewer).

**HandleUpdate()**

```cpp
virtual void HandleUpdate(ttxlaUpdateDesc_t* updateDescP) = 0
```

This method will be invoked whenever the HandleChange() method is called to process an update operation.

This method is not implemented in the TTXlaTableHandler base class. It must be provided by any classes derived from it, with appropriate logic to handle updated rows.

The previous version of the row that was updated from the table is available through the protected member row (of type TTXlaRowViewer). The new version of the row is available through the protected member row2 (also TTXlaRowViewer).

**generateSQL()**

```cpp
void generateSQL (ttxlaUpdateDesc_t* updateDescP, char* buffer, SQLINTEGER maxByteLen, SQLINTEGER* actualByteLenP)
```

This method can be used to print out the SQL associated with a given XLA record. The SQL string is returned through the buffer parameter. Allocate space for the buffer and specify its maximum length, maxByteLen. The actualByteLenP parameter returns information about the actual length of the SQL string returned.

If maxByteLen is less than the generated SQL string, a TTStatus error will be thrown, and the contents of buffer and actualByteLenP will not be modified.

**TTXlaTableList**

The TTXlaTableList class provides a list of TTXlaTableHandler objects and is used to dispatch update notification events to the appropriate TTXlaTableHandler object. When an update notification is received from XLA, the appropriate HandleXXXXX() method of the appropriate TTXlaTableHandler object is called to process the record.

For example, if an object of type CustomerTableHandler is handling changes to table CUSTOMER, and an object of type OrderTableHandler is handling changes to table ORDERS, the application should include both of these objects in a TTXlaTableList object. As XLA update notification records are fetched from XLA, they can be dispatched to the correct handler by simply calling TTXlaTableList::HandleChange().

The constructor has two forms:

```cpp
TTXlaTableList(TTXlaPersistConnection* cP, unsigned int num_tbls_to_monitor)
```

Where num_tbls_to_monitor is the number of database objects to monitor.

Or:

```cpp
TTXlaTableList(TTXlaPersistConnection* cP);
```
Where \( cP \) references the database connection to be used for XLA operations. This form of the constructor can monitor up to 150 database objects.

Usage
By registering `TTXlaTableHandler` objects in a `TTXlaTableList` object, the process of fetching update notification records from XLA and dispatching them to the appropriate methods for processing can be accomplished using a very simple loop.

Public members
None

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add()</code></td>
<td>Adds a <code>TTXlaTableHandler</code> object to the list.</td>
</tr>
<tr>
<td><code>del()</code></td>
<td>Deletes a <code>TTXlaTableHandler</code> object from the list.</td>
</tr>
<tr>
<td><code>HandleChange()</code></td>
<td>Processes a record obtained from a <code>ttXlaUpdateDesc_t</code> structure.</td>
</tr>
</tbody>
</table>

- `add()`: 
  ```
  void add(TTXlaTableHandler* tblh)
  ```
  Adds a `TTXlaTableHandler` object to the list.

- `del()`: 
  ```
  void del(TTXlaTableHandler* tblh)
  ```
  Deletes a `TTXlaTableHandler` object from the list.

- `HandleChange()`:
  ```
  void HandleChange(ttXlaUpdateDesc_t* updateDesc)
  ```
  Processes a record obtained from a `ttXlaUpdateDesc_t` structure.

When a `ttXlaUpdateDesc_t` object is received from XLA, it can be processed by calling this method, which determines which table the record references and calls the `HandleChange()` method of the appropriate `TTXlaTableHandler` object.

See "HandleChange()" on page 3-51 for `TTXlaTableHandler` objects, including a discussion of the `pData` parameter.

**TTXlaTable**

The `TTXlaTable` class encapsulates the metadata for a table being monitored for changes. It acts as a metadata interface for the TimesTen `ttXlaTblDesc_t` C data structure. (See "ttXlaTblDesc_t" in Oracle TimesTen In-Memory Database C Developer’s Guide.)

Usage
When a user application creates a class that extends `TTXlaTableHandler`, it will typically call `TTXlaTable::getColNumber()` to map a column name to its XLA column number. You can then use the column number as input to the
XLA classes

TTXlaRowViewer::Get() method. This is shown in the xlasubscriber2 demo in the TimesTen Quick Start. (Refer to ‘About the TimesTen TTClasses demos’ on page 1-6.)

This class also provides useful metadata functions to return the name, owner, and number of columns in the table.

Public members
None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColNumber()</td>
<td>Returns the column number of a column specified by name.</td>
</tr>
<tr>
<td>getNCols()</td>
<td>Returns the number of columns in the table.</td>
</tr>
<tr>
<td>getOwnerName()</td>
<td>Returns the name of owner of the table.</td>
</tr>
<tr>
<td>getTableName()</td>
<td>Returns the name of the table.</td>
</tr>
</tbody>
</table>

getColNumber()

int getColNumber(const char* colNameP) const

For a specified column name in the table, this method returns its column number, or -1 if there is no column by that name.

getNCols()

int getNCols() const

Returns the number of columns in the table.

getOwnerName()

const char* getOwnerName() const

Returns the database user name of the owner of the table.

getTableName()

const char* getTableName() const

Returns the name of the table.

TTXlaColumn

A TTXlaColumn object contains the metadata for a single column of a table being monitored for changes. It acts as a metadata interface for the TimesTen ttXlaColDesc_t C data structure. (See “ttXlaColDesc_t” in Oracle TimesTen In-Memory Database C Developer’s Guide.) Information including the column name, type, precision, and scale can be retrieved.

Usage

Applications can access a TTXlaColumn object by using the TTXlaRowViewer::getColumn() method.
XLA classes

Class Descriptions

Public members
None.

Public methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColName()</td>
<td>Returns the name of the column.</td>
</tr>
<tr>
<td>getPrecision()</td>
<td>Returns the precision of the column.</td>
</tr>
<tr>
<td>getScale()</td>
<td>Returns the scale of the column.</td>
</tr>
<tr>
<td>getSize()</td>
<td>Returns the size of the column data, in bytes.</td>
</tr>
<tr>
<td>getType()</td>
<td>Returns the data type of the column, as an integer.</td>
</tr>
<tr>
<td>getSysColNum()</td>
<td>Returns the system-generated column number of this column as stored in the database.</td>
</tr>
<tr>
<td>getUserColNum()</td>
<td>Returns 0 or a column number optionally specified by the user.</td>
</tr>
<tr>
<td>isNullable()</td>
<td>Indicates whether data in the column is allowed to have a NULL value.</td>
</tr>
<tr>
<td>isPKColumn()</td>
<td>Indicates whether this column is the primary key for the table.</td>
</tr>
<tr>
<td>isTTTimestamp()</td>
<td>Indicates whether this column is a TT_TIMESTAMP column.</td>
</tr>
<tr>
<td>isUpdated()</td>
<td>Indicates whether this column was updated.</td>
</tr>
</tbody>
</table>

getColName()

const char* getColName() const

Returns the name of the column.

getPrecision()

SQLULEN getPrecision() const

Returns the precision for data in the column, referring to the maximum number of digits that are used by the data type.

getScale()

int getScale() const

Returns the scale for data in the column, referring to the maximum number of digits to the right of the decimal point.

getSize()

SQLUINTEGER getSize() const

Returns the size of the column data, in bytes.

getType()

int getType() const

Returns an integer representing the ODBC SQL data type of the column.
XLA classes

**getSysColNum()**
```cpp
SQLINT32 getSysColNum() const
```
This is the system-generated column number of the column, numbered from 1. It is the same as the corresponding COLNUM value in SYS.COLUMNS. (See "SYS.COLUMNS" in Oracle TimesTen In-Memory Database System Tables and Limits Reference.)

**getUserColNum()**
```cpp
SQLINT32 getUserColNum() const
```
Returns 0 or a column number optionally specified by the user through the `ttSetUserColumnID` TimesTen built-in procedure. (See "ttSetUserColumnID" in Oracle TimesTen In-Memory Database Reference.)

**isNullble()**
```cpp
bool isNullable() const
```
Returns TRUE if data in the column is allowed to have NULL values, or FALSE otherwise.

**isPKColumn()**
```cpp
bool isPKColumn() const
```
Returns TRUE if this column is the primary key for the table, or FALSE otherwise.

**isTTTimestamp()**
```cpp
bool isTTTimestamp() const
```
Returns TRUE if this column is a TT_TIMESTAMP column, or FALSE otherwise.

**isUpdated()**
```cpp
bool isUpdated() const
```
Returns TRUE if this column was updated, or FALSE otherwise.
A
access control, 2-6
connections, 2-6
in XLA, 2-20
TimesTen features, 2-4
AUTOCOMMIT
connection, 3-10

B
batch operations, 3-27
binding parameters, 2-7
BindParameter method (batch), 3-28
duplicate parameters in SQL, 2-12
IN OUT parameters, 2-10
IN parameters, 2-8
OUT parameters, 2-10
registering parameters, 2-9
setParam() method, 3-19
BindParameter method, 3-28

C
catalog class, 3-33
client/server mode
compiling, 1-2
client/server, deferred prepare, 3-16
Close method, 3-17
column metadata, 3-38
Commit method, 3-8
compiler macros
TT_64BIT, 1-5
TTC_USE_STRINGSTREAM, 1-4
TTDEBUG, 1-5
USE_OLD_CPP_STREAMS, 1-5
compiling
client/server mode, 1-2
HP-UX, 1-5
Connect method, 3-7, 3-12
driver managers
example, 2-3
column
access control, 2-6
AUTOCOMMIT, 3-10
Connect method implementation, 2-3
connection
access control, 2-6
connection class definition, 2-2
Connect method definition, 2-4
DSNs, 2-5
pooling, 3-11
TTConnection class, 3-6

D
data types
using TTCmd, 3-22, 3-24, 3-28
setParam(), 3-19
using TTXlaRowViewer, 3-47
deferring, generating information, 1-5
deferring prepare, client/server, 3-16
Disconnect method, 3-8
disconnecting, 2-5
Disconnect method definition, 2-4
DML returning (RETURNING INTO), 2-10
DO_NOT_THROW flag (TTStatus), suppress exceptions, 3-5
driver managers, restrictions with, 1-5
Drop method, 3-17
DSNs (data source names), 2-5
duplicate parameter binding, 2-12
duplicateBindMode general connection attribute, 2-12
durableCommit method, 3-9
duplicateBindMode

E
error reporting
suppress exceptions, DO_NOT_THROW flag, 3-3
try/catch with TTStatus, 3-3
TTStatus class, 3-3
Execute method, 3-14, 3-16
ExecuteBatch method, 3-30
ExecuteImmediate method, 3-14, 3-16

F
FetchNext method, 3-17

G
gcc

Index
using to compile, 1-5
Get method, 3-47
getColumn method, 3-22
getColumnLength method, 3-24
getColumnName method, 3-26
getColumnType method, 3-26
getHdbc method, 3-9
getMaxRows method, 3-18
getNColumns method, 3-26
getNextColumn method, 3-24
getNextColumnNullable method, 3-24
getNParameters method, 3-25
getParam method, 3-21
getParamNullability method, 3-25
getParamPrecision method, 3-26
getParamScale method, 3-26
getQueryThreshold method, 3-18
getRowCount method, 3-19
GetTTContext method, 3-11

H
HDDBC abstraction class, 3-6
HP-UX
compling on, 1-5
HSTMT abstraction class, 3-14

I
IN OUT parameters, 2-10
IN parameters, 2-8
index metadata, 3-39
install
TTClasses library, 1-2
I/O streams
C++ flag settings, 1-4
isBeingExecuted method, 3-27
isConnected method, 3-8

L
logging, 2-16, 3-1
malloc

M
macros
64-bit TimesTen, 1-5
C++ streams, 1-4
gcc, 1-5
HP-UX, 1-5
TT_LIB64BIT, 1-5
TT_USE_STRINGSTREAM, 1-4
TTClasses compiler, 1-4

O
ODBC HDBC handle, 3-6
OUT parameters, 2-10

P
parameter binding, 2-7
duplicate parameters in SQL, 2-12
IN OUT parameters, 2-10
IN parameters, 2-8
OUT parameters, 2-10
registering parameters, 2-9
Prepare method, 3-16
PrepareBatch method, 3-27
printColumn method, 3-25

Q
query timeouts and thresholds, 2-16
queryBeingExecuted method (now isBeingExecuted), 3-27

R
REF CURSORS, 2-13, 3-21
registerParam method, 2-9
RePrepare method, 3-16
RETURNING INTO clause, 2-10
Rollback method, 3-8
rowids, ROWID type, 2-15

S
SetAutoCommitOff method, 3-10
SetAutoCommitOn method, 3-10
SetIsoReadCommitted method, 3-9
SetIsoSerializable method, 3-9
setMaxRows method, 3-18
setParam() method, 3-19
setParamLength method, 3-29
setParamNull method, 3-21, 3-30
setPrefetchCloseOff method, 3-10
setPrefetchCloseOn method, 3-10
setPrefetchCount method, 3-10
setQueryThreshold method, 3-18
setQueryTimeout method, 3-18
SQLBindParams ODBC function, 3-27
SQLError ODBC function, 3-3
statement class, 3-14

TTDEBUG, 1-5
USE_OLD_CPP_STREAMS, 1-5
make
debug libraries, 1-2
delete libraries and files, 1-2
optimized libraries, 1-2
shared debug library, 1-2
shared optimized library, 1-2
static debug library, 1-2
multithreaded applications, 3-11

static optimized library, 1-2
static optimized library, 1-2
stream, C++ flag settings, 1-4
stream, C++ flag settings, 1-4
table metadata, 3-36
table metadata, 3-36
threshold, SQL statements, 2-16
threshold, SQL statements, 2-16
timeout, SQL statements, 2-16
transaction
transaction
acknowledge XLA updates, 2-17
acknowledge XLA updates, 2-17
boundaries, 2-17
boundaries, 2-17
timeout, SQL statements, 2-16
log API
log API
overview, 2-17
overview, 2-17
TTCatalog class, 3-33
TTCatalog class, 3-33
TTCatalogIndex class, 3-39
TTCatalogIndex class, 3-39
TTCatalogTable class, 3-36
TTCatalogTable class, 3-36
TTCatalog class, 3-33
TTCatalog class, 3-33
TTClasses
TTClasses
compiler macros, 1-4
compiler macros, 1-4
compiling for client/server mode, 1-2
compiling for client/server mode, 1-2
compiling on UNIX, 1-1
compiling on UNIX, 1-1
compiling on Windows, 1-3
compiling on Windows, 1-3
definition, 2-1
definition, 2-1
install library, 1-2
install library, 1-2
logging, 2-16
logging, 2-16
logging levels, 3-2
logging levels, 3-2
TTCmd class, 3-14, 3-17, 3-22
TTCmd class, 3-14, 3-17, 3-22
TTCmd procedure, 3-16
TTCmd procedure, 3-16
ttCompact, ttCompactTS procedures, 3-9
ttCompact, ttCompactTS procedures, 3-9
TTCatalog class, 3-33
TTCatalog class, 3-33
TTCatalogIndex class, 3-39
TTCatalogIndex class, 3-39
TTCatalogTable class, 3-36
TTCatalogTable class, 3-36
TTCmd class, 3-14, 3-17, 3-22
TTCmd class, 3-14, 3-17, 3-22
TTCmd procedure, 3-16
TTCmd procedure, 3-16
TTConnection class, 3-6, 3-11
TTConnection class, 3-6, 3-11
TTConnectionPool class, 3-11
TTConnectionPool class, 3-11
TTClockWait procedure, 3-9
TTClockWait procedure, 3-9
TTDurableCommit procedure, 3-9
TTDurableCommit procedure, 3-9
TTError class, 3-4, 3-8
TTError class, 3-4, 3-8
TTGlobal class, 3-1
TTGlobal class, 3-1
TTStatus class (error reporting), 3-3, 3-8, 3-17
TTStatus class (error reporting), 3-3, 3-8, 3-17
TTStatus procedure, 3-11
TTStatus procedure, 3-11
TTWarning class, 3-4, 3-8
TTWarning class, 3-4, 3-8
TTXlaPersistConnection function, 3-42
TTXlaPersistConnection function, 3-42
TTXlaRowViewer class, 3-46
TTXlaRowViewer class, 3-46
TTXlaTableHandler class, 3-49
TTXlaTableHandler class, 3-49
TTXlaTableList class, 3-52
TTXlaTableList class, 3-52
ttXlaUpdateDesc_t data structure, 3-45
ttXlaUpdateDesc_t data structure, 3-45
UNIX
UNIX
install TTClasses library, 1-2
install TTClasses library, 1-2
USE_OLD_CPP_STREAMS, 1-5
USE_OLD_CPP_STREAMS, 1-5
Windows
Windows
compiling, 1-3
compiling, 1-3
XLA
XLA
access control, 2-20
access control, 2-20

Index-3