

# Matisse<sup>®</sup> C++ Programmer's Guide

August 2009



Matisse C++ Programmer's Guide

Copyright ©1992–2009 Matisse Software, Inc. All Rights Reserved.

Matisse Software, Inc.  
930 San Marcos Circle  
Burlingame, CA 94043  
USA

Printed in USA.

This manual and the software described in it are copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or in part, without prior written consent of Matisse Software, Inc. This manual and the software described in it are provided under the terms of a license between Matisse Software, Inc. and the recipient, and their use is subject to the terms of that license.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. and international patents.

TRADEMARKS: MATISSE and the MATISSE logo are registered trademarks of Matisse Software, Inc. All other trademarks belong to their respective owners.

PDF generated 21 August 2009

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
	Scope of This Document	5
	Before Reading This Document	5
	Additional Documentation for the C++ Binding	5
<b>2</b>	<b>Instructions for C++ Examples</b>	<b>6</b>
	Before Running the Examples	6
	Compiling the Examples	6
	Compatibility with Borland C++ Builder	6
	Generating Documentation with Doc++	7
<b>3</b>	<b>Connecting</b>	<b>8</b>
	Transaction (Connect.cpp)	8
	Read-Only Access (VersionConnect.cpp)	9
	Version Access (VersionNavigation.cpp)	10
	Other Options (AdvancedConnect.cpp)	12
<b>4</b>	<b>Working with Objects</b>	<b>15</b>
	Running ObjectsExample	15
	ObjectsExample.cpp Source	15
<b>5</b>	<b>Working with Values</b>	<b>18</b>
	Running ValuesExample	18
	ValuesExample.cpp Source	18
<b>6</b>	<b>Working with Relationships</b>	<b>22</b>
	Running RelationshipsExample	22
	RelationshipsExample.cpp Source	22
<b>7</b>	<b>Working with Indexes</b>	<b>26</b>
	Running IndexExample	26
	IndexExample.cpp Source	26
<b>8</b>	<b>Working with Entry-Point Dictionaries</b>	<b>29</b>
	Running EPDictExample	29
	EPDictExample.cpp Source	29
<b>9</b>	<b>Working with SQL</b>	<b>32</b>
	Running SQLExample	32
	SQLExample.cpp Source	32
<b>10</b>	<b>Optimization</b>	<b>36</b>
	Installing Sample Applications	36
	Creating Multiple Objects	36
	load.cpp Source	36

Other Operations on Multiple Objects. . . . .	38
delete.cpp Source. . . . .	39
Eliminating Instances from the Client Cache . . . . .	42
Clearing the C++ Instance Cache . . . . .	42
Using Pointers . . . . .	43
<b>11 Additional Topics . . . . .</b>	<b>44</b>
Arrays . . . . .	44
Namespaces. . . . .	44
Error Handling . . . . .	45
newman1.cpp Source. . . . .	45
newman2.cpp Source. . . . .	48
<b>Appendix A: Example Schema . . . . .</b>	<b>51</b>
<b>Appendix B: Generated Methods . . . . .</b>	<b>52</b>

# 1 Introduction

## Scope of This Document

This document is intended to help C++ programmers learn the aspects of Matisse design and programming that are unique to the Matisse C++ binding.

Aspects of Matisse programming that the C++ binding shares with other interfaces, such as basic concepts and schema design, are covered in *Getting Started with Matisse*.

Future releases of this document will add more advanced topics. If there is anything you would like to see added, or if you have any questions about or corrections to this document, please send e-mail to [support@matisse.com](mailto:support@matisse.com).

## Before Reading This Document

Throughout this document, we presume that you already know the basics of C++ programming and either relational or object-oriented database design, and that you have read the relevant sections of *Getting Started with Matisse*.

## Additional Documentation for the C++ Binding

*Getting Started with Matisse* and the sample code and example applications discussed in this document are available for download at:

<http://www.matisse.com/developers/documentation/>

The HTML-format *Matisse C++ Binding API Reference* is installed with Matisse at:

`%MATISSE_HOME%/docs/cxx/api/index.html`

## 2 Instructions for C++ Examples

### Before Running the Examples

Before running this and the following examples, you must do the following:

- Install Matisse 8.0.2 or a later version.
- Install a C++ compiler. Makefiles are provided for use with GNU GCC, Microsoft Visual C++ 6.0, and Sun Forte C++. With other compilers, you will need to create your own makefiles.
- Set the MATISSE\_HOME environment variable to the top-level directory of the Matisse installation.

### Compiling the Examples

Three sets of makefiles are supplied for several platforms which can build the supplied applications from a command line tool. Each makefile will compile all sources and link all applications in the directory.

- With GNU GCC, run:

```
gmake -f Makefile.gcc
```

- With Microsoft Visual C++, run:

```
nmake /fMakefile.win32
```

If this fails with the error message, 'cl' is not recognized as an internal or external command, enable the compiler's command-line option by running the batch file VCVARS32.BAT, which you should find in the Visual C++ \bin directory.

- With Sun Forte C++, run:

```
make -f Makefile.sun
```

These makefiles will not set up any databases needed by the applications, but will generate any source required from the ODL file.

### Compatibility with Borland C++ Builder

Matisse is compatible with Borland C++ Builder 5.5.1 or higher.

First, use the COFF to OMF converter (version 1.0.0.74 or higher) included with C++ Builder to convert the Matisse library `matisse.lib` to a Borland-compatible format, for example:

```
coff2omf c:\matisse\lib\matisse.lib matisseBorland.lib
```

Then specify that converted library when compiling, for example:

```
bcc32 myProgram.cpp matisseBorland.lib
```

## Generating Documentation with Doc++

You can generate an API reference for a set of `mt_sdl`-generated C++ classes with Doc++, the open-source tool we use to generate the Matisse C++ binding API documentation. Open source and compiled versions for Linux and Solaris are available at:

<http://www.zib.de/Visual/software/doc++/>

Compiled versions for Windows and FreeBSD are available from:

<http://docpp.sourceforge.net/download.html>

<http://www.freebsd.org/cgi/ports.cgi?query=doc%2B%2B&stype=all>

## 3 Connecting

All interaction between client C++ applications and Matisse databases takes place within the context of transactions (either explicit or implicit) established by database connections, which are transient instances of the `MtDatabase` class. Once the connection is established, your C++ application may interact with the database using the schema-specific methods generated by `mt_sdl` (see [Generated Methods](#) on page 52). The following sample applications show a variety of ways of connecting with a Matisse database.

### Transaction (Connect.cpp)

The following code connects to a database, starts and commits a transaction, and closes the connection:

```

/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 */

// the default Matisse C++ header
#include "matisseCXX.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// A simple program which opens a connection to a database to start a
// transaction.
int main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0] << " <host> <db>" << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        // open, select and start access to the database
        db.open();
        db.startTransaction();

        // read/write access
        std::cout << "Successful connection and open transaction to "
                  << db << std::endl;

        db.commit();
        db.close();
    }
    catch (MtException &e)
    {

```

```

    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}

```

## Read-Only Access (VersionConnect.cpp)

The following code connects to a database in read-only mode, suitable for reports:

```

/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 *
 */

// the default Matisse C++ header
#include "matisseCXX.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// A simple program that opens a read-only connection to a "snapshot" of the
// database at the time the connection is opened. For more information, see
// the "Version Access" section of Getting Started with Matisse.
int main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0] << " << <host> <db>" << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        // open, select and start access to the database
        db.open();
        db.startVersionAccess();

        // version connect implies read-only access
        std::cout << "Successful connection and version access to "
                  << db << std::endl;

        db.endVersionAccess();
        db.close();
    }
    catch (MtException &e)
    {

```

```
        std::cerr << e << std::endl;
    }
    catch (...)
    {
        std::cerr << "Unknown exception" << std::endl;
    }
    return 0;
}
```

## Version Access (VersionNavigation.cpp)

The following code illustrates methods of accessing various versions of a database.

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 *
 */

// the default Matisse C++ header
#include "matisseCXX.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// Shows how to create named versions when committing transactions and how to
// iterate through a list of versions. For additional information, see
// the "Named Versions" section of Getting Started with Matisse.
// define a local function in unnamed namespace
namespace {

// simple function which opens an iterator on the list of version names
void
listVersions(MtDatabase& db)
{
    MtStringIterator viter = db.versionIterator();

    std::cout << db << " has these versions:" << std::endl;
    while (viter.hasNext())
    {
        std::string vname = viter.next();
        std::cout << "\t" << vname << std::endl;
    }

    // close when finished.
    viter.close();
}

} // end anon namespace

int main(int ac, char **av)
{
```

```

if (ac < 3)
{
    std::cerr << "Usage: " << av[0] << " <host> <db>" << std::endl;
    return -1;
}
try
{
    MtDatabase db(av[1], av[2]);

    // open, select and start access to the database
    db.open();

    db.startTransaction();
    std::cout << "\nVersions before regular commit:" << std::endl;
    listVersions(db);
    db.commit();

    db.startTransaction();
    std::cout << "\nVersions after regular commit:" << std::endl;
    listVersions(db);

    // Providing a version name when committing a transaction
    // creates a named version of the database that must be
    // explicitly destroyed. The version name is the string
    // passed in argument (which distinguishes versions created
    // by this client) plus a unique ID number appended by the
    // server (which distinguishes this version from others
    // created by this client).
    std::string vername = db.commit("test");
    std::cout << "\nCommit to version named: " << vername << std::endl;

    db.startVersionAccess();
    std::cout << "\nVersions after named commit:" << std::endl;
    listVersions(db);
    db.endVersionAccess();

    // A saved version can be accessed read-only
    db.startVersionAccess(vername);
    std::cout << "\nSuccessful access within version: " << vername
    << std::endl;
    db.endVersionAccess();

    db.close();
}
catch (MtException &e)
{
    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}

```

## Other Options (AdvancedConnect.cpp)

This example shows how to enable the local client-server memory transport and to set or read various connection options and states.

```

/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 *
 */

// the default Matisse C++ header
#include "matisseCXX.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// A more complicated connection example which can be controlled by
// environment variables to specify additional access parameters, such
// as read-only access, user privileges, etc. NOTE: Security must be
// enabled for a database before users may be specified; for details,
// see the online help in mt_dba or section 5 of the Matisse Server
// Administration Guide.
class AdvancedConnect
{
private:
    MtDatabase db;

public:
    // constructor creates the MtDatabase
    AdvancedConnect(const std::string& host, const std::string& dbname)
        : db(host,dbname)
    {
    }

    void run()
    {
        if (getenv("MT_MEM_TRANS") != NULL)
            db.setOption(::MT_MEMORY_TRANSPORT, 1);

        // possible values are 0=RW, 1=RO, 2=DD (RW + Schema Modification)
        if (getenv("MT_DATA_ACCESS") != NULL)
        {
            int da_opt = atoi(getenv("MT_DATA_ACCESS"));
            db.setOption(::MT_DATA_ACCESS_MODE, da_opt);
        }

        if (getenv("dbuser") != NULL)
        {
            std::string user = getenv("dbuser") ? getenv("dbuser") : "";
            std::string passwd = getenv("dbpasswd") ? getenv("dbpasswd") : "";
            db.open(user, passwd);
        }
    }
}

```

```

else
{
    db.open();
}

start(isReadOnly());
printStats();

// do other work here ...

end();
db.close();
}

void start(bool readonly)
{
    if (readonly)
        db.startVersionAccess();
    else
        db.startTransaction();
}

void end(void)
{
    if (db.isVersionAccessInProgress())
        db.endVersionAccess();
    else if (db.isTransactionInProgress())
        db.commit();
    else
        std::cerr << "No transaction/version access in progress"
                    << std::endl;
}

bool isMemoryTransportOn()
{
    return (db.getOption(::MT_TRANSPORT_TYPE) == ::MT_MEM_TRANSPORT);
}

bool isReadOnly()
{
    return (db.getOption(::MT_DATA_ACCESS_MODE) == 1);
}

void printState()
{
    if (!db.isConnectionOpen())
    {
        dbmsg("not connected");
    }
    else
    {
        if (db.isTransactionInProgress())
            dbmsg("read-write transaction underway");
        else if (db.isVersionAccessInProgress())
            dbmsg("read-only version access underway");
        else
            dbmsg("no transaction underway");
    }
    std::string msg = "MEMORY_TRANSPORT is ";

```

```
    msg += (isMemoryTransportOn() ? "on" : "off");
    dbmsg(msg);
    msg = "Access is ";
    msg += (isReadOnly() ? "readonly" : "readwrite");
    dbmsg(msg);
}

void dbmsg(const std::string& msg)
{
    std::cout << db << ": " << msg << std::endl;
}
};

int main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0] << " <host> <db>" << std::endl;
        return -1;
    }

    try {
        AdvancedConnect conn(av[1], av[2]);
        conn.run();
    }
    catch (MtException& mte)
    {
        std::cerr << mte << std::endl;
    }
    catch (...)
    {
        std::cerr << "Unknown exception" << std::endl;
    }
}
```

## 4 Working with Objects

### Running ObjectsExample

This sample program creates two objects (one Person and one Employee), lists all Person objects (which includes both objects, since Employee is a subclass of Person), deletes both objects, then lists all Person objects again to show the deletion. Note that because FirstName and LastName are not nullable, they *must* be set when creating an object.

1. Follow the instructions in [Before Running the Examples](#) on page 6.
2. Create and initialize a database as described in [Getting Started with Matisse](#).
3. Load objects.odl into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select objects.odl.

4. Generate C++ class files.

```
mt_sdl stubgen -cxx objects.odl
```

5. Compile and link the application with the appropriate makefile (see [Compiling the Examples](#) on page 6).
6. Optionally, generate the API reference for the schema (see [Generating Documentation with Doc++](#) on page 7). The following command assumes that the Doc++ executable is named docxx, that you want the output in HTML format, and that you want the output to go into a subdirectory named docs:

```
docxx -d docs -H *.h
```

7. Launch the application:

```
ObjectsExample host database
```

### ObjectsExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 *
 */

// the required Matisse C++ header
#include "matisseCXX.h"
// the headers for schema classes
#include "Person.h"
#include "Employee.h"

// the namespaces that contain the Matisse classes
```

```

using namespace matisse;
using namespace matisse::reflect;

// A simple example which shows how to use generated code for user classes.
int main(int ac, char **av)
{
    if (ac < 3)
    {
        cerr << "Usage: " << av[0] << " <host> <db>" << endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        // open, select and start access to the database
        db.open();
        db.startTransaction();

        // create a new Person
        Person& p = Person::create(db);
        // modify attributes
        p.setFirstName("John");
        p.setLastName("Smith");
        p.setAge(42);

        // create a new Employee
        Employee& e = Employee::create(db);
        // set attributes
        e.setFirstName("Jane");
        e.setLastName("Jones");
        // age is nullable, so leave unset

        // create timestamp from string representation
        matisse::MtTimestamp ts("2002-01-01");
        e.setHireDate(ts);
        matisse::MtNumeric num(1000000.00);
        e.setSalary(num);

        // list all Persons
        std::cout << std::endl << Person::getInstanceNumber(db)
            << " Persons in the database" << std::endl;
        // open an iterator on all the instances of Person and all
        // subclasses; if you wish to exclude subclasses, use
        // Person::ownInstanceIterator() instead
        MtObjectIterator<Person> piter = Person::instanceIterator(db);
        while (piter.hasNext())
        {
            // use object reference
            Person &x = piter.next();
            // show attributes and name of class
            std::cout << "\t" << x.getFirstName() << " " << x.getLastName()
                << " is a " << x.getMtClass().getMtName() << std::endl;
        }

        // remove created objects
        std::cout << "\nRemoving created objects" << std::endl;
        p.remove();
        e.remove();
    }
}

```

```
// list again to show deletion, similar to above
    std::cout << "\nAfter deletion:" << std::endl;
    std::cout << Person::getInstanceNumber(db)
        << " Persons in the database" << std::endl;
    MtObjectIterator<Person> piter2 = Person::instanceIterator(db);
    while (piter2.hasNext())
    {
        Person &x = piter2.next();
        std::cout << "\t" << x.getFirstName() << " " << x.getLastName()
            << " is a " << x.getMtClass().getMtName() << std::endl;
    }

    db.commit();
    db.close();
}
catch (MtException &e)
{
    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}
```

## 5 Working with Values

### Running ValuesExample

This example is generated by the makefile used in `ObjectsExample`, and it uses the same database. It creates an object, manipulates its values in various ways as described in the source-code comments, imports the data in the file `matisse.gif` to an attribute value, creates a new file from the stored data, then removes the object.

To launch the application:

```
ValuesExample host database
```

### ValuesExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 */

// the required Matisse C++ header
#include "matisseCXX.h"
// the headers for schema classes
#include "Person.h"
#include "Employee.h"
// doing some read/write to files
#include <fstream>

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// A more complete example using classes to modify Matisse objects.
// Uses an image file to stream data in/out of an attribute. Also shows
// how to catch a specific exception.
int main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0] << " << <host> <db>" << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        // open, select and start access to the database
        db.open();
        db.startTransaction();
    }
}
```

```

// create a new Employee
Employee& e = Employee::create(db);

// setting string attributes
e.setComment("setting values");
e.setFirstName("John");
e.setLastName("Jones");

// setting numbers
e.setAge(42);

// setting Date
matisse::MtTimestamp ts("2002-02-02");
e.setHireDate(ts);

// setting Numeric
matisse::MtNumeric num(2958.33);
e.setSalary(num);

// getting
std::cout << std::endl << e.getComment() << std::endl;
std::cout << "\tEmployee: " << e.getFirstName() << " "
    << e.getLastName() << std::endl;

// suppress output if no value set
// use generated isAgeNull() method to check if value is null
if (!(e.isAgeNull()))
    std::cout << "\t" << e.getAge() << " years old" << std::endl;
std::cout << "\tNumber of Dependents: " << e.getDependents()
    << std::endl;
std::cout << "\tSalary: $" << e.getSalary() << std::endl;
std::cout << "\tHired on: " << e.getHireDate() << std::endl;

// changing values (getting and setting)
e.setDependents(e.getDependents() + 2);
if (e.getFirstName() == "John")
{
    matisse::MtNumeric sal = e.getSalary();
    matisse::MtNumeric mul(1.05);
    // operations on matisse::MtNumeric supported
    sal *= mul;
    e.setSalary(sal);
}
e.setComment("changing values");

// Getting again to show effect of removing value
// similar to above
std::cout << std::endl << e.getComment() << std::endl;
std::cout << "\tEmployee: " << e.getFirstName() << " "
    << e.getLastName() << std::endl;
// use generated isAgeNull() method to check if value is null
if (!(e.isAgeNull()))
    std::cout << "\t" << e.getAge() << " years old" << std::endl;
std::cout << "\tNumber of Dependents: " << e.getDependents()
    << std::endl;
std::cout << "\tSalary: $" << e.getSalary() << std::endl;
std::cout << "\tHired on: " << e.getHireDate() << std::endl;

```

```

// Blob access:
// The Photo attribute's type is MtImage which is
// streamable. Use an array of bytes to read an image file from
// disk and store into the attribute.

std::cout << std::endl << "Accessing streamable attribute"
    << std::endl;

// setting blob
int bufSize = 15;    // small buff size for demo purposes
int actualBytes;
int totalBytes = 0;
// use auto_ptr to clean up when done
std::auto_ptr<char> tmpBytes(new char[bufSize]);
// get the attribute for streaming
MtAttribute &pAtt = e.getPhotoAttribute(db);

// binary file on disk
std::ifstream is("matisse.gif", std::ios::in | std::ios::binary);
if (!is)
{
    std::cerr << "matisse.gif does not exists" << std::endl;
}
else
{
    // initial call to truncate data
    e.setPhotoElements(::MtByte*)tmpBytes.get(), 0,
        MT_BEGIN_OFFSET, true);
    while (!is.eof())
    {
        actualBytes = (is.read(tmpBytes.get(), bufSize)).gcount();
        e.setPhotoElements(::MtByte*)tmpBytes.get(), actualBytes,
            MT_CURRENT_OFFSET, true);
        totalBytes += actualBytes;
    }
    is.close();
}
std::cout << "transferred " << totalBytes
    << " bytes from matisse.gif to object"
    << std::endl;

// read blob and store to file, just reverse the process from above
std::ofstream os("new.gif", std::ios::binary);
totalBytes = 0;
if (!os)
{
    std::cerr << "Failure opening new.gif" << std::endl;
}
else
{
    // reset the steam
    e.getPhotoElements(::MtByte*)tmpBytes.get(), 0, MT_BEGIN_OFFSET);
    do {
        actualBytes = e.getPhotoElements(::MtByte*)tmpBytes.get(),
            bufSize, MT_CURRENT_OFFSET);
        os.write(tmpBytes.get(), actualBytes);
        totalBytes += actualBytes;
    } while (actualBytes == bufSize);
    os.close();
}

```

```

    }
    std::cout << "transferred " << totalBytes
              << " bytes from object to new.gif"
              << std::endl;

    db.commit();

    // set read-only access and catch exception

    db.startVersionAccess();

    std::cout << "\nAttempting to modify " << e.getFirstName() + " "
              << e.getLastName() << " salary within read-only access"
              << std::endl;
    try
    {
        matisse::MtNumeric num(3958.40);
        e.setSalary(num);
    }
    catch (MtException& mte)
    {
        std::cout << mte;
        if (mte.getErrorCode() == MATISSE_NOTRANS)
            std::cout << "... as expected.." << std::endl;
        else
            std::cout << " but was not expected.." << std::endl;
    }

    db.endVersionAccess();

    // remove created object from database
    std::cout << "\nRemoving created object" << std::endl;
    db.startTransaction();
    e.remove();
    db.commit();

    db.close();
}
catch (MtException &e)
{
    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}

```

## 6 Working with Relationships

### Running RelationshipsExample

This example creates several objects, manipulates the relationships among them in various ways as described in the source-code comments, then removes the objects.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Create and initialize a database as described in *Getting Started with Matisse*.
3. Load `examples.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `examples.odl`.

4. Generate C++ class files.

```
mt_sdl stubgen -cxx examples.odl
```

5. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).

6. Optionally, generate the API reference for the schema (see *Generating Documentation with Doc++* on page 7). The following command assumes that the Doc++ executable is named `docxx`, that you want the output in HTML format, and that you want the output to go into a subdirectory named `docs`:

```
docxx -d docs -H *.h
```

7. Launch the application:

```
RelationshipsExample host database
```

### RelationshipsExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 *
 */

// the required Matisse C++ header
#include "matisseCXX.h"
// the headers for schema classes
#include "Manager.h"
#include "Employee.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;
```

```

// A simple application that demonstrates how to access, modify, and navigate
// relationships.
int main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0] << " <host> <db>" << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        matisse::MtTimestamp ts; // default is now
        matisse::MtNumeric num(48.3);

        db.open();
        db.startTransaction();

        // create a manager
        Manager& m1 = Manager::create(db);
        m1.setFirstName("J.C.");
        m1.setLastName("Dithers");
        m1.setHireDate(ts);
        m1.setSalary(num);
        // set the successor object for reportsTo (in this case it
        // refers to itself, i.e., this manager is the big boss at
        // the top of the reporting hierarchy)
        m1.setReportsTo(m1);

        // create another manager
        Manager& m2 = Manager::create(db);
        m2.setFirstName("Dagwood");
        m2.setLastName("Bumstead");
        m2.setHireDate(ts);
        num = 42.0; // MtNumeric supports operator=(double)
        m2.setSalary(num);
        m2.setReportsTo(m1); // the manager

        // create an employee
        Employee& e = Employee::create(db);
        e.setFirstName("Elmo");
        e.setLastName("Tuttle");
        ts = "2001-12-31"; // MtTimestamp supports operator=(char *)
        e.setHireDate(ts);
        num = 22.0;
        e.setSalary(num);
        e.setReportsTo(m2);

        // specify an assistant for each of the two managers
        m1.setAssistant(e);
        m2.setAssistant(e);

        // check the assistantOf from the employee
        std::cout << "\nInitial settings:" << std::endl;
        // get the entire list of successors (class Manager)
        MtObjectArray<Manager> list = e.getAssistantOf();
        // access list as an array using operator[]

```

```

for (int i=0; i < (int)list.size(); i++)
    std::cout << "\t" << e.getFirstName() << " is "
        << list[i].getFirstName() << "'s assistant" << std::endl;

// create a few extra persons
Person& c1 = Person::create(db);
c1.setFirstName("Alexander");
c1.setLastName("Bumstead");

Person& c2 = Person::create(db);
c2.setFirstName("Cookie");
c2.setLastName("Bumstead");

// a std::vector of Type* can be converted from
// MtObjectArray<T> which can be used as argument to set a list
// of successors
std::vector<Person*> cVec(2);
cVec[0] = &c1;
cVec[1] = &c2;
MtObjectArray<Person> children(db,cVec);
m2.setChildren(children);

// When accessing a relationship with single cardinality [1] or
// [0,1], the access method returns a pointer to the object so
// that it can be checked for NULL.
// in this case getFather() returns Person*
std::cout << "\nSet successors:" << std::endl;
std::cout << "\t" << c1.getFirstName() << " is "
    << (c1.getFather()->getFirstName() << "'s child"
    << std::endl;
std::cout << "\t" << c2.getFirstName() << " is "
    << (c2.getFather()->getFirstName() << "'s child"
    << std::endl;

// convenient use of C++ overloaded operators
std::cout << "\nCompare objects (use ==):" << std::endl;
std::cout << "\tDo " << c1.getFirstName() << " and "
    << c2.getFirstName() << " have the same father (==)? "
    << (c1.getFather() == c2.getFather()) << std::endl;

// the following code shows how to add and remove successors
Person& c3 = Person::create(db);
c3.setFirstName("Baby");
c3.setLastName("Bumstead");

// append to existing list
m2.appendChildren(c3);

std::cout << "\nAdd another successor:" << std::endl;
std::cout << "\tNow " << m2.getFirstName() << " has "
    << m2.getChildrenSize() << " children" << std::endl;

// use of MtObjectArray<> for specifying a list to remove
std::vector<Person*> rVec(2);
rVec[0] = &c2;
rVec[1] = &c3;
MtObjectArray<Person> children2(db, rVec);

```

```

// remove
m2.removeChildren(children2);

// another signature supports a single removal
// i.e. m2.removeChildren(c2);

std::cout << "\nRemove two successors:" << std::endl;
std::cout << "\tNow " << m2.getFirstName() << " has "
    << m2.getChildrenSize() << " children" << std::endl;

// clear all the successors
m2.clearChildren();
std::cout << "\nClear successors:" << std::endl;
std::cout << "\tNow " << m2.getFirstName() << " has "
    << m2.getChildrenSize() << " children" << std::endl;

// reset to original array
m2.setChildren(children);

// iterator
std::cout << "\nAdd successors and iterate through children.."
    << std::endl;
std::cout << "\t" << m2.getFirstName() << "'s children:" << std::endl;
MtObjectIterator<Person> piter = m2.childrenIterator();
while (piter.hasNext())
{
    Person &p = piter.next();
    std::cout << "\t\t" << p.getFirstName() << std::endl;
}

//remove created objects
m1.remove();
m2.remove();
e.remove();
c1.remove();
c2.remove();
c3.remove();

db.commit();
db.close();
}
catch (MtException &e)
{
    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}

```

## 7 Working with Indexes

### Running IndexExample

This example is generated by the makefile used in RelationshipsExample, and it uses the same database. It first creates some Person objects in the database and lists their names; then, using the PersonName index, checks whether the database contains an entry for a person matching the specified name; then deletes the objects.

To run the application:

```
IndexExample host database firstName lastName
```

### IndexExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 */

// the required Matisse C++ header
#include "matisseCXX.h"
#include "Person.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// Sample application using an index for lookup and sorted listing.
int main(int ac, char **av)
{
    if (ac < 5)
    {
        std::cerr << "Usage: " << av[0]
            << " <host> <db> <firstName> <lastName>"
            << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        db.open();

        // add some Person objects
        db.startTransaction();
        Person& p1 = Person::create(db);
        p1.setFirstName("John");
        p1.setLastName("Jones");
    }
}
```

```

Person& p2 = Person::create(db);
p2.setFirstName("Fred");
p2.setLastName("Jones");
Person& p3 = Person::create(db);
p3.setFirstName("John");
p3.setLastName("Murray");
Person& p4 = Person::create(db);
p4.setFirstName("Fred");
p4.setLastName("Flintstone");

std::cout << "\nAdding 4 objects" << std::endl;
db.commit();

db.startVersionAccess();

// just show the instances available
std::cout << "\nPersons available:" << std::endl;
MtObjectIterator<Person> pIter = Person::instanceIterator(db);
while (pIter.hasNext())
{
    Person &p = pIter.next();
    std::cout << "\t" << p.getFirstName() << " " << p.getLastName()
        << std::endl;
}

// search for a Person with specified first last name
std::string firstName = av[3];
std::string lastName = av[4];

std::cout << "\nLooking for : " << firstName << " " << lastName
    << std::endl;

// The lookup function must return a Person* to allow for NULL
// to represent no match
Person *found = Person::lookupPersonName(db, lastName, firstName);
if (found != NULL)
{
    std::cout << "\tfound " << found->getFirstName() << " "
        << found->getLastName() << std::endl;
}
else
{
    std::cout << "\tNobody found" << std::endl;
}

// instead of searching, open an iterator within the specified
// criteria; an Index can specify upto 4 criteria and this API
// would change accordingly (see examples.odl for specification)
std::string fromFirstName = "Fred";
std::string toFirstName = "John";
std::string fromLastName = "Jones";
std::string toLastName = "Murray";
std::cout << "\nIndex from \'" << fromFirstName << " "
    << fromLastName << "\' to \'" << toFirstName << " "
    << toLastName << "\'" << std::endl;
MtObjectIterator<Person> ppIter = Person::personNameIterator(db,
    fromLastName, fromFirstName, toLastName, toFirstName);

```

```
while (ppIter.hasNext())
{
    Person &p = ppIter.next();
    std::cout << "\t" << p.getFirstName() << " " << p.getLastName()
        << std::endl;
}
db.endVersionAccess();

db.startTransaction();

std::cout << "\nRemoving objects..." << std::endl;
p1.remove();
p2.remove();
p3.remove();
p4.remove();
db.commit();

db.close();
}
catch (MtException &e)
{
    std::cerr << e << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}
return 0;
}
```

## 8 Working with Entry-Point Dictionaries

### Running EPDictExample

This example is generated by the makefile used in RelationshipsExample, and it uses the same database. It first creates some Person objects in the database and lists them; then, using the commentDict entry-point dictionary, counts the number of objects with Comments fields containing the search string passed at the command line; then deletes the objects.

To run the application:

```
EPDictExample host database search_string
```

### EPDictExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 */

// the required Matisse C++ header
#include "matisseCXX.h"
// the schema class
#include "Person.h"

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;

// Sample application which uses an entry-point dictionary associated
// with the Person::comment attribute. The commentDict is a full text
// dictionary so every word in the comment attribute will provide an
// entry point. (see examples.odl for specification)
int main(int ac, char **av)
{
    if (ac < 4)
    {
        std::cerr << "Usage: " << av[0] << " <host> <db> <searchString>"
            << std::endl;
        return -1;
    }
    try
    {
        MtDatabase db(av[1], av[2]);

        db.open();

        // add some Person objects
        db.startTransaction();
```

```

Person& p1 = Person::create(db);
p1.setFirstName("John");
p1.setLastName("Jones");
p1.setComment("weak needs have always hindered reality");
Person& p2 = Person::create(db);
p2.setFirstName("Fred");
p2.setLastName("Jones");
p2.setComment("in reality weak knees hindered many");

std::cout << "\nAdding 2 objects with comments:" << std::endl
  << " " << p1.getFirstName() << " " << p1.getLastName()
  << " says \" " << p1.getComment() << "\" " << std::endl
  << " " << p2.getFirstName() << " " << p2.getLastName()
  << " says \" " << p2.getComment() << "\" " << std::endl;

db.commit();

db.startVersionAccess();

// set the search string from command line
std::string searchString = av[3];
int hits = 0;

std::cout << "\nLooking for : " << searchString << std::endl;
// open an iterator on the number of Persons that match
MtObjectIterator<Person> pIter = Person::commentDictIterator(db,
  searchString);
while (pIter.hasNext())
{
  Person &p = pIter.next();
  std::cout << "\t" << p.getFirstName() << " " << p.getLastName()
    << std::endl;
  hits++;
}
std::cout << db << ": " << hits << " comment fields contained \" "
  << searchString << "\" " << std::endl;

db.endVersionAccess();

db.startTransaction();

std::cout << "\nRemoving objects..." << std::endl;
p1.remove();
p2.remove();
db.commit();

db.close();
}
catch (MtException &e)
{
  std::cerr << e << std::endl;
}
catch (...)
{
  std::cerr << "Unknown exception" << std::endl;
}

```

```
return 0;  
}
```

## 9 Working with SQL

### Running SQLExample

This example executes two SQL queries and displays their results.

The first query (`select name, id, boss.name from Employee where id > 2`) uses standard SQL syntax and returns “column” (attribute/relationship) and “row” (object) names and attribute values in the familiar table format.

The second query (`select Ref(Employee), Ref(boss) from Employee where id > 2`) uses Matisse’s object extensions and returns object IDs (OIDs), which in turn are used to get the names and values.

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Create and initialize a database as described in *Getting Started with Matisse*.
3. Load `sql_eg.odl` into the database. From the Enterprise Manager, select your database and right click on ‘Schema->Import ODL Schema’, then select `sql_eg.odl`.

4. Generate C++ class files.

```
mt_sdl stubgen -cxx sql_eg.odl
```

5. Load the sample data into the database. From the Enterprise Manager, select your database and right click on ‘Schema->Import ODL Schema’, then select `sql_eg.sql`. This will make the SQL statement appear in the Query Editor window. Then click ‘Execute Query’.
6. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).

7. Launch the application:

```
SQLExample host database
```

### SQLExample.cpp Source

```
/*
 * Copyright (c) 1992-2002 Matisse Software Inc. All Rights Reserved.
 *
 * This file (in both binary and source code form) is subject to the
 * Supplemental License Terms governing use, modification and redistribution
 * of Sample Code accompanying the Matisse(R) software.
 */

// the default Matisse C++ header
#include "matisseCXX.h"
// the local schema header
#include "sql_eg.h"

// the namespaces that contain the Matisse classes
```

```

using namespace matisse;
using namespace matisse::reflect;
using namespace matisse::sql;

// Sample application which shows how to access data via SQL
// queries. Shows both the traditional form of accessing individual
// attribute values (columns) as well as getting OIDs which map to
// C++ instances. Database must be loaded before running this
// application; use the supplied sql_eg.sql list of SQL commands.
class SQLExample
{
private:
    MtDatabase db;

public:
    // simple construction to open and connect to database
    SQLExample(const std::string& host, const std::string& dbname)
        :db(host,dbname)
    {
        db.open();
        db.startVersionAccess();
        std::cout << "Accessing: " << db << std::endl << std::endl;
    }

    // end and close
    ~SQLExample()
    {
        db.endVersionAccess();
        db.close();
    }

    // send traditional query
    void doItPlain()
    {
        // create a statement, execute some sql and iterate through row/cols
        MtStatement stmt(db);
        std::string sqlStr =
            "select name, id, boss.name from Employee where id > 2";
        std::cout << "Query: " << sqlStr << std::endl;

        MtResultSet res = stmt.executeQuery(sqlStr);

        // list names and types
        // column index is 1-based
        for (int i = 1; i <= res.getColumnCount(); i++)
        {
            std::cout << "column " << i << " " << res.getColumnName(i)
                << " type is: "
                << res.getColumnTypeName(i) << std::endl;
        }
        // since we know that we asked for string and integer, we can dump
        // all the results
        // use the next method to move through rows
        while (res.next())
        {
            std::cout << "Employee name: " << res.getString(1) << ", id: "
                << res.getInteger(2) << ", boss : " << res.getString(3)
                << std::endl;
        }
    }
}

```

```

    // always remember to close the statement when done
    stmt.close();
}

// query for objects
void doItWithObjects()
{
    // to get a list of objects, use the Ref operator
    MtStatement stmt(db);
    std::string sqlStr =
        "select Ref(Employee), Ref(boss) from Employee where id > 2;";
    std::cout << "Query: " << sqlStr << std::endl;

    MtResultSet res = stmt.executeQuery(sqlStr);

    // list names and types
    // column index is 1-based
    for (int i = 1; i <= res.getColumnCount(); i++)
    {
        std::cout << "column " << i << " " << res.getColumn(i) << res.getColumnName(i)
            << " type is: "
            << res.getColumnTypeName(i) << std::endl;
    }
    // still use the ResultSet, but get the object and coerce to the
    // class we expect (note: using dynamic_cast<> will throw
    // std::bad_cast if the result type is not acceptable)
    while (res.next())
    {
        // show the class name of the reference
        std::cout << "[class = "
            << res.getMtObject(1)->getMtClass().getMtName()
            << ", ";

        // Note: using dynamic_cast with pointer conversion returns
        // NULL on failure. If you would prefer an exception
        // (std::bad_cast), use object reference to make it an
        // assertion, e.g.:
        // Employee* e = &dynamic_cast<Employee*>*(res.getMtObject(1));
        Employee *e = dynamic_cast<Employee *>(res.getMtObject(1));
        Manager *m = dynamic_cast<Manager *>(res.getMtObject(2));
        if (e == NULL)
        {
            std::cout << "Empty Employee from Query!" << std::endl;
            continue;
        }
        std::cout << "Employee name: " << e->getName()
            << ", id: " << e->getId() << " boss: "
            << ((m != NULL) ? m->getName() : "[no boss]")
            << std::endl;
    }

    // remember to the close statement when done
    stmt.close();
}
};

// main is simply a way to get args to the SQLExample class and catch
// any exceptions
int main(int ac, char **av)

```

```
{
  if (ac < 3)
  {
    std::cerr << "Usage: " << av[0] << " <host> <db>" << std::endl;
    return -1;
  }

  try {
    SQLExample sqleg(av[1], av[2]);
    std::cout << "Traditional SQL Query Use..." << std::endl;
    sqleg.doItPlain();
    std::cout << std::endl;
    std::cout << "Getting Objects from SQL Query..." << std::endl;
    sqleg.doItWithObjects();
  }
  catch (MtSQLException& sqle)
  {
    std::cerr << "MtSQL Exception: " << sqle << std::endl;
  }
  catch (MtException& mte)
  {
    std::cerr << "Matisse exception: " << mte << std::endl;
  }
  catch (std::exception& e)
  {
    std::cerr << "Standard exception: " << e.what() << std::endl;
  }
  catch (...)
  {
    std::cerr << "Unknown exception" << std::endl;
  }

  return 0;
}
```

# 10 Optimization

## Installing Sample Applications

To install the sample applications for this section and the next:

1. Follow the instructions in *Before Running the Examples* on page 6.
2. Create and initialize a database as described in *Getting Started with Matisse*.
3. Load `company.odl` into the database. From the Enterprise Manager, select your database and right click on 'Schema->Import ODL Schema', then select `company.odl`.

4. Generate C++ class files.

```
mt_sdl stubgen -cxx company.odl
```

5. Compile and link the application with the appropriate makefile (see *Compiling the Examples* on page 6).

## Creating Multiple Objects

When an application needs to create several objects, it is more efficient to have the server allocate multiple objects in one call rather than one at a time inside of a loop. To accomplish this, use the `MtClass::createInstances(num)` method, which returns an `MtObjectArray<>`. For example, to create `count` instances of the class `Employee`:

```
MtObjectArray<Employee> emps(Employee::getClass(db)).createInstances(count).typeless();
for (int i = 0; i < count; i++)
{
    Employee &e = emps[i];
    // set attributes for employee i
    e.setFirstName(...);
    // etc
}
// commit (or rollback) still required
// etc
```

Note that `Employee::getClass(db)` is a generated static method for `Employee` which returns an instance of `MtClass`. Since the `MtClass::createInstances()` method returns an `MtObjectArray<MtObject>` (an array of generic `MtObject`), it must be converted to typeless form to be used in the copy constructor for `MtObjectArray<Employee>`; this is accomplished by the `MtObjectArray<>::typeless()` method.

For sample code illustrating this technique, see `load.cpp`. See the source for instructions on running the application.

## load.cpp Source

```
#include "matisseCXX.h"
```

```

#include "company.h"
#include <iostream>
#include <sstream>
#include <iomanip>

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace std;

// Sample application demonstrating the most efficient way to allocate a
// large number of objects in one transaction: create all the instances
// in a single server call, use this array of objects to set individual
// attributes, then commit. The drawback is that if commit fails (for
// example, due to a non-unique ID), no instances are created.
//
// Usage
// load <hostname> <database> will load 100 Employees starting at id#0
// additional arguments for number of objects to create and starting
// ID are optional.
//
// For each employee created a name is generated based on ID: first
// name is Annnnnn, lastname = Bnnnnnn where nnnnnn is id number padded
// with 0's for width=6. ID is guaranteed to be unique because the
// unique_key option is set (see attribute declaration in company.odl).

int
main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0]
            << " <host> <db> <#create>? <#startID>?"
            << endl;
        return -1;
    }

    try
    {
        MtDatabase db(av[1], av[2]);

        int count = 100;
        int idstart = 0;

        if (ac > 3)
            count = ::atoi(av[3]);
        if (ac > 4)
            idstart = ::atoi(av[4]);

        db.open();
        db.startTransaction();

        std::cout << "Creating " << count << " Employees" << std::endl;

        // create an array of new instances
        MtObjectArray<Employee> emps(
            (Employee::getClass(db)).createInstances(count).typeless());
        for (int i = 0; i < count; i++)
        {
            Employee &e = emps[i];

```

```

    std::ostringstream os1;
    os1 << "A" << std::setw(6) << std::setfill('0') << i + idstart;

    std::ostringstream os2;
    os2 << "B" << std::setw(6) << std::setfill('0') << i + idstart;

    e.setFirstName(os1.str());
    e.setLastName(os2.str());
    e.setId(i + idstart);
}
std::cout << "Done" << std::endl;

db.commit();
db.close();

}
catch (MtException& mte)
{
    std::cerr << "Matisse exception: " << mte << std::endl;
}
catch (std::exception& e)
{
    std::cerr << "Standard exception: " << e.what() << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}

return 0;
}

```

When creating a large quantity of objects of different classes, an application can still optimize client-server traffic by preallocating OIDs. A block of OIDs can be preallocated for object creation within a transaction; whenever an object is created, it will use a preallocated OID if available, avoiding traffic to the server. Note, however, that these OIDs are “wasted” if not used by the current transaction. See the Matisse C++ binding API reference for complete documentation of `MtDatabase::preallocate(int num)`.

## Other Operations on Multiple Objects

Generally speaking, data transferred between client and server is optimized by the client cache to avoid unnecessary round trips. For example, when an object instance is first accessed, all the basic attribute information (not including relationships or streamable attributes) for that instance is transferred to the client as well, and this information will stay in the cache during the transaction. With this in mind, when an application needs to access several objects in succession, it can optimize the data transferred between client and server by preloading all the instances into the client cache with a single server operation, so that each instance access will not require a separate server access. This is accomplished using the `load()` method on an `MtObjectArray<>`. For example, to access all the `Employee` instances which are successors to a particular `Department` object's team relationship:

```
MtObjectArray<Employee> emps = aDepartment.getTeam(); // get the array of successors
```

```

emps.load(); // load all the Employee instances
for (int i = 0; i < emps.size(); i++)
{
    Employee &e = emps[i];
    myid = e.getId(); // collect info from Employee instance
}

```

As a convenience, there is also an `MtObjectArray<>::remove()` method which can be used to remove all the instances without having to write a similar loop calling `e.remove()`.

Note that the creation of the `MtObjectArray<Employee>` does not populate the client cache, nor does it create C++ object instances. It only retrieves an array of OID (unique object identifiers). The cache and instances are affected only by access.

For sample code illustrating this technique, see `delete.cpp`. See the source for instructions on running the application.

## delete.cpp Source

```

#include "matisseCXX.h"
#include "company.h"
#include <iostream>
#include <sstream>
#include <iomanip>

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace std;

// How to delete objects?
// Often it is easiest just to use SQL, e.g. DELETE * FROM EMPLOYEE
// or DELETE * FROM EMPLOYEE WHERE ...", but it can be accomplished
// in client code.

// This application can be used in several ways to delete all the
// objects in the database or a range of objects.
//
// 1) Delete all the objects: just pass the host and database args.
// The objects are deleted in groups of 100 by iterating through the
// instances, then the entire array removed at once.
//
// 2) Delete a single object: pass host, database, and Employee ID.
//
// 3) Delete a range of objects: pass host, database, and the
// first and last Employee IDs in the range to be deleted. In this
// case, we loop-preload the entire array of objects into the client
// cache, then remove them individually.
//
// 4) Delete alternate objects in a range of objects, starting with
// the first: pass the same four arguments as for (3), plus a fifth
// argument (the value is not checked, so any other value will work).
//
// The code for each of these four alternatives is optimized for the
// particular task.
//
// NOTE: Before running delete, run load to generate sample data.

```

```

int
main(int ac, char **av)
{
    if (ac < 3)
    {
        std::cerr << "Usage: " << av[0]
            << " <host> <db> <idmin>? <idmax>? <trim>?"
            << endl;
        return -1;
    }

    try
    {
        MtDatabase db(av[1], av[2]);

        db.open();
        db.startTransaction();

        if (ac == 3) // simplest case, delete them all
        {
            int delCount = 100;
            std::cout << "Deleting all employees. (" << delCount
                << " at a time)" << std::endl;

            // get the iterator
            MtObjectIterator<Employee> eiter = Employee::instanceIterator(db);

            // An even more straightforward approach would be to use
            // Employee::getInstanceNumber(db) but that could possibly
            // create a very large (and unnecessary) array if instance
            // count was very high.
            while (eiter.hasNext())
            {
                MtObjectArray<Employee> emps(eiter.next(delCount));

                std::cout << " removing " << emps.size() << " Employees"
                    << std::endl;
                emps.remove();
            }
            eiter.close();
        }
        else
        {
            // will be deleting a range
            bool trim = false;
            int fromId = ::atoi(av[3]);
            int toId = fromId;
            if (ac > 4)
                toId = ::atoi(av[4]);

            // existence of arg is indication to delete alternately
            if (ac > 5)
                trim = true;

            std::cout << "Deleting employees " << fromId << " to " << toId
                << (trim ? " (alternating)" : "")
                << std::endl;
        }
    }
}

```

```

// get the iterator for entire range
MtObjectIterator<Employee> eiter = Employee::employeeIdIterator(db,
    fromId, toId);

// quick check to see if we have any instances to deal with
if (eiter.hasNext())
{
    if (trim) // going to trim the list to delete
    {
        // get the whole array
        MtObjectArray<Employee> emps0(eiter.next(toId - fromId + 1));
        const MtOid *oids = emps0.oids();
        int oidCount = 0;

        // use auto ptr for safe cleanup
        auto_ptr<MtOid> tmpOids(new MtOid[emps0.size() / 2 + 1]);

        for (int j = 0; j < emps0.size(); j++)
        {
            if ((j % 2) == 0)
                *(tmpOids.get() + oidCount++) = oids[j];
        }

        // now create new version of object array
        MtObjectArray<Employee> emps(db, oidCount, tmpOids.get());
        std::cout << "removing " << emps.size() << " Employees"
            << std::endl;

        // could use MtObjectArray<>::remove as below, but
        // looping to check values, so load whole array first
        emps.load();
        int minId = emps[0].getId();
        int maxId = minId;
        for (int i = 0; i < emps.size(); i++)
        {
            if (emps[i].getId() < minId)
                minId = emps[i].getId();
            else if (emps[i].getId() > maxId)
                maxId = emps[i].getId();
            emps[i].remove();
        }
        std::cout << " min ID = " << minId
            << ", max ID = " << maxId
            << std::endl;
    }
    else // not trimming, delete the specified range
    {
        // since we know id is unique, this is worst case
        int count = toId - fromId + 1;
        // grab the whole array
        MtObjectArray<Employee> emps(eiter.next(count));
        std::cout << "removing " << emps.size() << " Employees"
            << std::endl;

        // could use MtObjectArray<>::remove as below, but
        // looping to check values, so load whole array first
        emps.load();
    }
}

```

```

        int minId = emps[0].getId();
        int maxId = minId;
        for (int i = 0; i < emps.size(); i++)
        {
            if (emps[i].getId() < minId)
                minId = emps[i].getId();
            else if (emps[i].getId() > maxId)
                maxId = emps[i].getId();
            emps[i].remove();
        }
        std::cout << " min ID = " << minId
                  << ", max ID = " << maxId
                  << std::endl;
    }
}

db.commit();
db.close();

} // end of try
catch (MtException& mte)
{
    std::cerr << "Matisse exception: " << mte << std::endl;
}
catch (std::exception& e)
{
    std::cerr << "Standard exception: " << e.what() << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}

return 0;
}

```

## Eliminating Instances from the Client Cache

`MtObjectArray<>::unload` can be used to remove objects from the client cache (both those that were loaded explicitly with `load()` and those loaded automatically by access). For example, if a version access (or transaction) is used for continued access to large numbers of objects, it can unload the objects from the Matisse client cache after access is complete.

## Clearing the C++ Instance Cache

A cache of all the C++ instances the C++ binding creates is maintained by `MtDatabase`. A C++ instance is a very small “stub” to a Matisse instance consisting only of references to the relevant OID and `MtDatabase` (plus any C++ overhead).

The C++ instance cache is populated automatically during object lookup. There is no explicit load method. The cache can be configured to be cleared automatically, for example at connection or transaction boundary. This policy can be set by `MtDatabase::setObjectCacheBoundary()`. Since these C++ stubs are very small, automatic clearing is usually adequate, but when necessary (for example, during a lengthy version access) the cache can be cleared explicitly by calling `MtDatabase::clearObjectCache()`. See the *Matisse C++ Binding API Reference* for additional discussion of these two methods.

The default clearing policy set during creation of a new `MtDatabase` is `NO_BOUND`, which means that the cache can only be cleared explicitly; either by a call to `clearObjectCache()` or as would happen during object cleanup when the `MtDatabase` is destroyed (in the `MtDatabase` destructor).

In either case (automatic or explicit), the C++ instance cache is cleared completely. There is no method for removing only selected C++ instances.

The client cache and C++ instance cache operate independently. That is, a C++ instance may be cached when the corresponding Matisse instance is not, and vice-versa.

## Using Pointers

Within the binding, pointers to C++ instances are only used in a few cases where existence needs to be confirmed (for example, during an index lookup). All other cases use a C++ reference; this design helps avoid accumulation of dangling pointers to cache instances that no longer exist. However, use of pointers cannot be strictly enforced as it is only a design pattern, therefore, use caution when using pointers. For example, when the C++ instance cache clearing policy is `TIME_BOUND` (transaction), do not hold pointers across transaction boundaries.

# 11 Additional Topics

## Arrays

The C++ binding defines two template classes to be used by API methods which pass or return arrays.

- `MtObjectArray<>` is used only for arrays of Matisse objects and provides an API which includes methods to access the individual objects using the `[]` operator as well as methods supporting conversions to/from `std::vector`. See [Other Operations on Multiple Objects](#) on page 38 for more information about `MtObjectArray<>` and `delete.cpp` for examples.
- `MtArray<>` is used for arrays of any type which can be used as an attribute value, both C++ primitives as well as Matisse-supplied types such as `MtTimestamp`. The `MtArray<>` class also provides element access via the `[]` operator, but additionally, it is derived from `MtValue` and therefore inherits methods to access the type, size, and so on. For example:

```
// a Department has a list of integers in the Days attribute
MtArray<int> days = thisDepartment.getDays();
for (int i = 0; i < days.getSize(); i++)
{
    std::cout << i << " : " << days[i] << std::endl;
}

// make a new array and set 'Days' attribute
// previously allocated array, 'myDays' to use..
int *myDays = ....; // can be allocated or stack based
// make new int array and let it borrow the pointer
MtArray<int> newDays = newIntegers(mySize, myDays, MtPointerBorrow);
for (i = 0; i < newDays.getSize(); i++)
    newDays[i] = ...;
thisDepartment.setDays(newDays);
```

See the *Matisse C++ Binding API Reference* (<http://%MATISSEHOME%/docs/cxx/api/index.html>) for the full API and documentation on `MtValue` and `MtArray` instance creation.

## Namespaces

The C++ binding defines and uses the `matisse` namespace; all core classes for the binding are defined under the space. Any user generated code, by default, contains no namespace specification and is therefore in the top level (`::`). If the `-n package` option is passed to `mt_sdl`, the code will be generated with the namespace specification for `package`. The namespace specification affects how the classes are scoped as well as the search path used by the default object factory (`MtPackageObjectFactory`) to create the proper C++ class based on the database object class.

If a class is in a namespace, then the header will put all definitions within a namespace `<name> { }` and in order to use that class in user applications, the class name must be fully qualified unless the module contains a `using namespace <name>`. For example, to refer to the `MtDatabase` class in a source file without a `using` clause, it would be referred to as `matisse::MtDatabase`.

For more information on namespaces, refer to a C++ reference such as *The C++ Programming Language* by Bjarne Stroustrup.

## Error Handling

Example applications `newman1.cpp` and `newman2.cpp` demonstrate how to break a complex update up into a series of short transactions so that any exceptions resulting from invalid data will not roll back valid data. See the comments in `newman1.cpp` for more information and instructions on running the applications. See [Installing Sample Applications](#) on page 36 for installation instructions.

### newman1.cpp Source

```
#include "matisseCXX.h"
#include "company.h"
#include <iostream>
#include <sstream>
#include <iomanip>

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace std;

// Sample application demonstrates adding relationship successors one at
// a time in separate transactions in order to handle any exceptions
// individually. The advantage of this approach is that exceptions will
// will roll back only the transactions involving invalid data and other
// transactions will commit successfully. (If all the updates were
// attempted in the same transaction, one exception would result in all
// being rolled back.)
//
// Compare newman2.cpp to see the differences when using SQL to
// accomplish the same results
//
// *** Usage ***
//
// NOTE: Before running newman1, run load to generate sample data for
// the Employee class.
//
// newman1 <host> <db> <asstFirstName> <asstLastName> <count> <id>
// will create <count> managers starting with ID number <id>, each with
// the assistant Employee whose first and last name are specified.
//
// Use the <id> option to generate exceptions deliberately. For example:
//
// newman1 <host> <database> A000000 B000000 1 2000
// newman1 <host> <database> A000000 B000000 2 2000
//
// Since the first command creates a manager with the ID 2000, the
// second command's attempt to create another with the same ID will
// fail and be rolled back.

int
main(int ac, char **av)
```

```

{
int actualCreateCount = 0, actualTransCount = 0, actualAbortCount = 0;
int newManagerCount = 50; // default

if (ac < 5)
{
std::cerr << "Usage: " << av[0]
<< " <host> <db> <asstFirstName> <asstLastName> <count>? <id>?"
<< std::endl;
return -1;
}

try
{
MtDatabase db(av[1], av[2]);
int useId = 1000; // default starting id for manager

std::string asstFirstName = av[3];
std::string asstLastName = av[4];

if (ac > 5)
newManagerCount = ::atoi(av[5]);
if (ac > 6)
useId = ::atoi(av[6]);

db.open();

std::cout << "Creating " << newManagerCount
<< " managers with an assistant of "
<< asstFirstName << " " << asstLastName << std::endl;
for (int i = 0; i < newManagerCount; i++)
{
try
{
actualTransCount++;
db.startTransaction();

// find the employee by name lookup; in this simple
// example, this lookup should be moved outside the
// loop, since the search criteria do not change, and
// left here for example only as we use the logic of
// finding Employee to commit/rollback transaction.
Employee *e =
dynamic_cast<Employee*>(Person::lookupPersonName(db,
asstLastName, asstFirstName));

if (e != NULL)
{
// create a new manager
Manager &m = Manager::create(db);
std::ostringstream os1;
os1 << "A" << std::setw(6) << setfill('0') << useId;

std::ostringstream os2;
os2 << "B" << std::setw(6) << setfill('0') << useId;

// manager attributes
m.setFirstName(os1.str());
m.setLastName(os2.str());
}
}
}
}
}

```

```

        m.setId(useId);
        // set the assistant
        m.setAssistant(*e);

        db.commit();
        actualCreateCount++;
    }
    else
    {
        std::cerr << "could not find " << asstFirstName << " "
            << asstLastName << std::endl;
        if (db.isTransactionInProgress())
        {
            db.rollback();
        }
        actualAbortCount++;
    }
}
// an exception during create will rollback transaction
catch (MtException& mte)
{
    if (db.isTransactionInProgress())
    {
        db.rollback();
    }
    actualAbortCount++;
    std::cerr << mte << std::endl;
    std::cerr << "At transaction #: " << i
        << ", id: " << useId << std::endl;
}
useId++;    // update for the next employee id
}
db.close();

}
catch (MtException& mte)
{
    std::cerr << mte << std::endl;
}
catch (std::exception& e)
{
    std::cerr << "Standard exception: " << e.what() << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}

std::cout << "Summary: " << actualCreateCount
    << " managers created in " << actualTransCount
    << " transactions, with " << actualAbortCount << " rollbacks"
    << std::endl;

return 0;
}

```

## newman2.cpp Source

```

#include "matisseCXX.h"
#include "company.h"
#include <iostream>
#include <sstream>
#include <iomanip>

// the namespaces that contain the Matisse classes
using namespace matisse;
using namespace matisse::reflect;
using namespace matisse::sql;
using namespace std;

// Same as newman1.cpp, except using SQL.
//
// Usage
// See notes in newman1.cpp.

int
main(int ac, char **av)
{
    int actualCreateCount = 0, actualTransCount = 0, actualAbortCount = 0;
    int newManagerCount = 50; // default

    if (ac < 5)
    {
        std::cerr << "Usage: " << av[0]
            << " <host> <db> <asstFirstName> <asstLastName> <count>? <id>?"
            << std::endl;
        return -1;
    }

    try
    {
        MtDatabase db(av[1], av[2]);

        db.open();

        int useId = 999;

        std::string asstFirstName = av[3];
        std::string asstLastName = av[4];

        if (ac > 5)
            newManagerCount = ::atoi(av[5]);
        if (ac > 6)
            useId = ::atoi(av[6]);

        std::cout << "Creating " << newManagerCount
            << " managers with an assistant of "
            << asstFirstName << " " << asstLastName << std::endl;

        for (int i = 0; i < newManagerCount; i++)
        {
            try
            {

```

```

actualTransCount++;
db.startTransaction();

MtStatement stmt(db);
std::string selName = "sel1"; // selection name for
    // setting relationship

// similar to non-SQL version, finding the employee
// could be outside the loop, since it is static
std::ostringstream sqs_emps;
sqs_emps
    << "select Ref(Employee) from Employee where firstName = "
    << asstFirstName << " and lastName = " << asstLastName
    << " into " << selName;

stmt.execute(sqs_emps.str());

#if 0
// check
MtResultSet res =
    stmt.executeQuery("select Ref(s) from sel1 s");
if (res.next())
{
    Employee *e = dynamic_cast<Employee*>(res.getMtObject(1));
    std::cout << e->getFirstName() << " " << e->getLastName()
        << std::endl;
}
else
{
    std::cerr << "could not find " << asstFirstName << " "
        << asstLastName << std::endl;
}
#endif

std::ostringstream sqs_man;

// buld the insert command
sqs_man << "insert into Manager(firstName, lastName, id, assistant) Values("
    << "A" << std::setfill('0') << std::setw(6)
    << useId << " ,"
    << "B" << std::setfill('0') << std::setw(6)
    << useId << " ,"
    << useId << " ," << selName << ")";

int res = stmt.executeUpdate(sqs_man.str());
//     std::cout << "Update: " << sqs_man.str() << std::endl
//     << "Result : " << res << std::endl;

if (res != 1)
{
    // if insert command didn't have an exception,
    // result would be 1, so we should not reach this
    std::cerr << "Should not be here!" << std::endl;
    // but let continue
}
db.commit();
actualCreateCount++;
}
catch (MtSQLException &mtsql)

```

```
{
    std::cerr << mtsqle << std::endl;
    if (db.isTransactionInProgress())
    {
        db.rollback();
    }
    actualAbortCount++;
}
catch (MtException &mte)
{
    std::cerr << mte << std::endl;
    if (db.isTransactionInProgress())
    {
        db.rollback();
    }
    actualAbortCount++;
}
useId++;
}

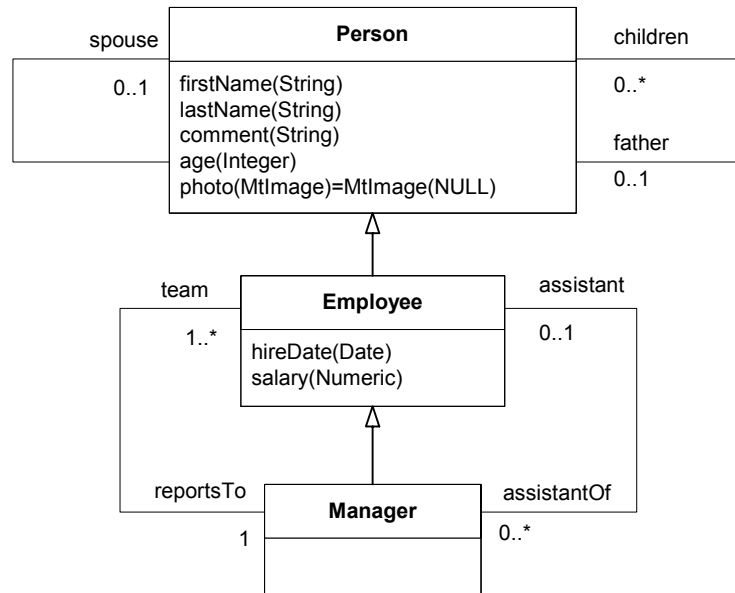
db.close();

}
catch (MtException& mte)
{
    std::cerr << mte << std::endl;
}
catch (std::exception& e)
{
    std::cerr << "Standard exception: " << e.what() << std::endl;
}
catch (...)
{
    std::cerr << "Unknown exception" << std::endl;
}

std::cout << "Summary: " << actualCreateCount
    << " managers created in " << actualTransCount
    << " transactions, with " << actualAbortCount << " rollbacks"
    << std::endl;

return 0;
}
```

# Appendix A: Example Schema



```

interface Person : persistent
{
  attribute String firstName;
  attribute String lastName;
  attribute String Nullable comment;
  attribute Integer Nullable age;
  attribute Image Nullable photo = NULL;
  relationship Person spouse[0,1] inverse Person::spouse;
  readonly relationship Person father[0,1] inverse Person::children;
  relationship List<Person> children inverse Person::father;
  mt_index personName
  criteria {person::lastName MT_ASCEND 16},
    {person::firstName MT_ASCEND 16};
  mt_entry_point_dictionary commentDict entry_point_of comment
  make_entry_function "make-full-text-entry";
};

interface Employee : Person : persistent
{
  attribute Date hireDate;
  attribute Numeric salary;
  readonly relationship List<Manager> assistantOf inverse Manager::assistant;
  relationship Manager reportsTo inverse Manager::team;
};

interface Manager : Employee : persistent
{
  relationship List<Employee> team[1,-1] inverse Employee::reportsTo;
  relationship Employee assistant[0,1] inverse Employee::assistantOf;
};
  
```

## Appendix B: Generated Methods

The following methods are defined in the C++ class files generated by `mt_sdl`. Definitions are in `class.h`, inlines in `class.hpp`, and other source in `class.cpp`.

### For schema classes

The following methods are created for each schema class. These are class methods (also called static methods): that is, they apply to the class as a whole, not to individual instances of the class. These examples are taken from `Person`.

Count instances	<code>getInstanceNumber(const MtDatabase &amp;db)</code>
Open an iterator	<code>MtObjectIterator&lt;Person&gt; instanceIterator(const MtDatabase &amp;db)</code>
Sample constructor	<code>Person &amp;create(const MtDatabase &amp;db)</code>
Sample ostream << operator	<code>std::ostream &amp;operator&lt;&lt;(std::ostream &amp;o, const Person &amp;obj)</code> This non-class method overloads the insertion operator for streams, which enables code such as: <pre>cout &lt;&lt; aPerson &lt;&lt; endl</pre>
Get descriptor	<code>MtClass&amp; getClass(const MtDatabase &amp;db)</code> Returns an <code>MtClass</code> object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.
Factory constructor	<code>MtObject* newStub(const MtDatabase &amp;db, ::MtOid oid)</code> This constructor is called by <code>MtObjectFactory</code> . It is public for technical reasons but is not intended to be called directly by user methods.

### For attribute descriptors

The following methods are created for each attribute descriptor. For example, if the ODL definition for class `Check` contains attribute descriptors `Date` and `Amount`, the `Check.h` file will contain the methods `getDate` and `getAmount`. This and following examples are taken from `Person::firstName`.

#### For all attribute descriptors

Remove value `removeFirstName()`

#### For scalar (non-list-type) attribute descriptors only

Get value `getFirstName()`

Set value `setFirstName(const std::string &val)`

**Get descriptor** `getFirstNameAttribute(const MtDatabase &db)`

Returns an `MtAttribute` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

### For list-type attribute descriptors only

The following methods are created for each list-type attribute descriptor. These examples are from `Person::photo`.

**Get value** `MtArray<unsigned char> getPhoto()`

**Set value** `setPhoto(const MtArray<unsigned char> &val)`

**Get elements** `getPhotoElements(::MtByte *value, unsigned int len, unsigned int offset=MT_CURRENT_OFFSET)`

**Set elements** `setPhotoElements(::MtByte *value, unsigned int len, unsigned int offset=MT_CURRENT_OFFSET, bool discardAfter=MT_FALSE)`

**Count elements** `getPhotoSize()`

**Get descriptor** `MtAttribute& getPhotoAttribute(const MtDatabase &db)`

Returns an `MtAttribute` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

### For all relationship descriptors

The following methods are created for each relationship descriptor. These examples are from `Person::spouse`.

**Clear successors** `clearSpouse()`

**Get descriptor** `MtRelationship& getSpouseRelationship(const MtDatabase &db)`

Returns an `MtRelationship` object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

### For relationship descriptors where the maximum cardinality is 1

The following methods are created for each relationship descriptor with a maximum cardinality of 1. These examples are from `Manager::assistant`.

**Get successor** `Employee* getAssistant()`

**Set successor** `setAssistant(const Employee &succ)`

### For relationship descriptors where the maximum cardinality is greater than 1

The following methods are created for each relationship descriptor with a maximum cardinality greater than 1. These examples are from `Manager::team`.

**Get successors** `MtObjectArray<Employee> getTeam()`

Open an iterator	MtObjectIterator<Employee> teamIterator()
Count successors	getTeamSize()
Set successors	setTeam(const MtObjectArray<Employee> &succs)
Add successors	Insert one successor before any existing successors: prependTeam(const Employee &succ)
	Add one successor after any existing successors: appendTeamp(const Employee &succ)
	Add multiple successors after any existing successors: appendTeam(const MtObjectArray<Employee> &succs)
Remove successors	removeTeam(const Employee &succ) removeTeam(const MtObjectArray<Employee> &succs)
	Remove specified successors.

## For index descriptors

The following methods are created for every index defined for a database. These examples are for the only index defined in the example, `Person::personName`. The number of attributes in the lookup and iterator methods is dependent on the number of criteria defined for the index (in this case, two, `lastName` and `firstName`).

Lookup	Person* lookupPersonName(const MtDatabase &db, const std::string & lastName, const std::string & firstName)
Open an iterator	MtObjectIterator<Person> personNameIterator(const MtDatabase &db, const std::string & fromLastName, const std::string & fromFirstName, const std::string & toLastName, const std::string & toFirstName, const MtClass* filterClass=NULL, ::MtDirection direction=MT_DIRECT, int numObjPerBuffer=MT_MAX_PREFETCHING)
Get descriptor	MtIndex& getPersonNameIndex(const MtDatabase &db)
	Returns an <code>MtIndex</code> object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.

## For entry-point descriptors

The following methods are created for every entry-point dictionary defined for a database. These examples are for the only dictionary defined in the example, `Person::commentDict`.

Lookup	Person* lookupCommentDict(const MtDatabase &db, const std::string &value)
Open an iterator	MtObjectIterator<Person> commentDictIterator(const MtDatabase &db, const std::string &value, const MtClass* filterClass=NULL, int numObjPerBuffer=MT_MAX_PREFETCHING)
Get descriptor	MtEntryPointDictionary& getCommentDictDictionary(const MtDatabase &db)
	Returns an <code>MtEntryPointDictionary</code> object. This method supports advanced Matisse programming techniques such as dynamically modifying the schema.