VisualAge® C++ Professional for AIX®

IBM® Open Class™: Application Control

Version 5.0

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

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1998, 2000

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 1998, 2000

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

® (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. ® Copyright IBM Corp. 1998, 2000. All rights reserved.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program’s tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architechture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

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

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered tradmark of Ziff Communicatoins Company and is used by IBM Corporation under license

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

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

Chapter 1. Open Class Threading Model

A typical application is a *single-threaded* process. The path of execution enters the main() function, runs the application and library methods, and terminates as it exits main(). Within a process, the thread runs your code.

However, many threads may execute simultaneously within a single process, independent to each other and to the main thread. Processes running more than one thread are *multi-threaded*. Multiple processes running concurrently may also be multi-threaded.

The operating systems that support IBM Open Class are themselves multi-threaded. Therefore multi-threaded applications can take advantage of hardware platforms that have multiple CPU devices. Threaded Open Class applications can achieve tremendous performance gains over the default, single-threaded, applications.

You must apply multiple threads programmatically. Each thread can run your application's global methods and objects' methods. Sometimes two or more threads are calling methods on the same object simultaneously. Several threads may simultaneously call the same method or multiple methods that access the same data belonging to the application object. When multiple threads contend for access to the same data, you must *synchronize*, or *serialize*, their access.

To protect application data from simultaneous access by multiple threads of execution, the Open Class library provides *mutex locks*, referred to as *resources*, and *event locks*, referred to as *conditions*. Open Class classes and/or member functions that are themselves protected from simultaneous access to multiple threads of execution are *thread-safe*. Open Class classes and member functions are thread-safe unless documented otherwise.

Open Class application objects typically use thread objects; unlike objects in other threading models, they do not derive from a thread object or implement a threading interface.

GUI and Non-GUI Threads
Open Class has two concrete base thread classes:

- INonGUIThread
- IThread

INonGUIThread is the basic thread class; however, IThread, which is derived from INonGUIThread, offers additional threaded behavior for graphical user interface situations. To start a thread within a process, you must explicitly use an object of either INonGUIThread or IThread.

These two thread classes have no application state of their own. They share access to the application objects' data.

RELATED CONCEPTS

© Copyright IBM Corp. 1998, 2000

"Behavior of IBM Open Class Threads"
"Resources and Conditions" on page 3
"Thread-Specific Data" on page 4
"Thread Scheduling" on page 5

RELATED TASKS

"Start a Thread" on page 6

Behavior of IBM Open Class Threads

The behavior of a multi-threaded process is non-deterministic; you do not know the following:

- When a thread may run a member function.
- How many threads may be running the same member function concurrently.
- How many threads may be concurrently running member functions that access the same data.

When C++ data is only accessible through member functions, synchronizing the access to those member functions synchronizes access to the data. To ensure that no more than one-thread-at-a-time is modifying the same data or that data is not being read by one thread while another is modifying it, you can either do the following:

- Bracket the accessing code with a static resource lock that can only be owned by a single thread at a time.
- Protect the accessing code with a condition that blocks all threads from the accessing code until that condition triggers either one or all the waiting threads to access the code.

It is recommended that you not synchronize IThread objects, because they are graphical user interface (GUI) threads. If IThread objects block, or sleep, they might delay the response of the graphical user interface, depending upon the operating system. For extended operations by an IThread object, that object should launch an INonGUIThread object which can be synchronized instead.

The Thread Currently Running an Application Object's Member Function
For GUI applications, call the IThread::current member function to obtain a reference to the ICurrentThread object that represents the currently executing thread. For non-GUI applications, call INonGUIThread::current to obtain a reference to the ICurrentNonGUIThread object that represents the currently executing thread.

The ICurrentThread class represents the current thread of execution for a GUI application. The IBM Open Class allows only a single object of this class. The ICurrentThread class lets you do the following:

- Call application member functions that only the current thread of execution can invoke
- Set/query attributes for the currently executing thread
- Start event processing
- Suspend the current thread until another thread has terminated

The ICurrentNonGUIThread class represents the current thread of execution for a non-GUI application. The IBM Open Class allows only a single object of this class.

The ICurrentNonGUIThread has the functionality of ICurrentThread without the GUI support, such as the ability to start event processing.

RELATED CONCEPTS

"Chapter 1. Open Class Threading Model" on page 1
"Resources and Conditions"
"Thread-Specific Data" on page 4
"Thread Scheduling" on page 5

RELATED TASKS

"Start a Thread" on page 6

Resources and Conditions

Resource Locking
An IResource object is a lock you can use to serialize thread access to code. Typically, a class declares a single static IResource object that is shared by all the objects of this class. IResource is an abstract base class. IResource objects are instances of either subclass:

- IPrivateResource (synchronizing within the same process)
- ISharedResource (synchronizing among concurrent processes)

An application object must have a lock in every public and protected method that accesses the same data. You would not need to put a lock in private methods, as long as all the public/protected methods which call that private method have locks in their body that precede the call. The following method is structured for the first thread entering the method's body to lock the static resource, thereby blocking any other threads from calling this code until the owning thread exits this method:

```
class myClass
{
   IPrivateResource myLock;
   myClass::myLockedMethod()
   {
      IResourceLock mySetLock(myLock);
      // locking for the remainder of
      // myLockedMethod occurs here
      // ...
      // single-threaded access assured for the remainder
      // of myLockedMethod
   }
   // mySetLock is deleted and myLock is unlocked when
   // myLockedMethod goes out of scope
}
```

The IResourceLock object implicitly calls the functions IPrivateResource::lock and IPrivateResource::unlock. Open Class automatically calls the IResourceLock destructor if either the thread exits the bracketing method or if an exception is thrown. IResourceLock saves you from explicitly calling its destructor or from using try/catch blocks for exceptions which depend upon destroying the IResourceLock object.

Event Monitoring
Class ICondition monitors for an event. An ICondition object blocks all threads

from access to the subsequent code. The ICondition object can *signal* the first waiting thread to run the protected code, or it can *broadcast* to unblock all of the waiting threads.

RELATED CONCEPTS

"Chapter 1. Open Class Threading Model" on page 1
"Behavior of IBM Open Class Threads" on page 2
"Thread-Specific Data"
"Thread Scheduling" on page 5

RELATED TASKS

"Start a Thread" on page 6

Thread-Specific Data

Per-thread data never needs to be synchronized. You can create thread-specific data (per-thread instance data) using the following steps:

1. Derive your application object from IThreadFn.
2. Create the data variables you want within your derived class.
3. Implement the virtual IThreadFn::run() method to use that data.

Application objects derived from IThreadFn must also use an INonGUIThread or IThread object to start a thread's execution.

Open Class provides another technique for creating per-thread data. You can use the template class IThreadLocalStorage to create per-thread global variables that may be needed by a library or application.These variable serve as thread-specific global pointers. An IThreadLocalStorage object cannot be instanced by new(), nor can be declared by any object or within the main() function. An IThreadLocalStorage object must be declared globally.

An IThreadLocalStorage object can access a single type of data, which will have an initial value of 0. If you want many types of data referenced by an IThreadLocalStorage object, create a helper class that contains all the desired data types, and then pass it to the IThreadLocalStorage template construction. For optimal performance, do not create a separate IThreadLocalStorage object for every type of data you want this thread to store.

In previous versions of Open Class, IThread handled thread-local storage (which is still supported). However, IThreadLocalStorage improves the implementation.

RELATED CONCEPTS

"Behavior of IBM Open Class Threads" on page 2
"Resources and Conditions" on page 3
"Chapter 1. Open Class Threading Model" on page 1"Thread Scheduling" on page 5

RELATED TASKS

"Start a Thread" on page 6

Thread Scheduling

Often an application creates more threads than the hardware had processors. You can designate some threads to be run before considering running other threads by assigning relative priorities to the threads; this is how you can schedule the multi-threaded execution. In previous versions of Open Class, thread scheduling was based on the OS/2® model (which is still supported). However, new scheduling application code is portable to Windows NT® and AIX as well as OS/2.

Open Class prioritizes execution using a priority class and a priority level. In the OS/2 model, both processes and threads had their own priority classes and their own priority levels. Priority classes were specified with an enum, and priority levels were specified with integers (available in the OS/2 system's scheduling).

Scheduling in this release uses the Windows NT model: processes can only have a priority class, and threads can only have a priority level. Thus, the process automatically sets the priority class for all of its threads, but each thread can set/reset its own priority within the process. Both process priorities and thread priorities are specified as enumerated types.

Process scheduling uses the following enum:

enum INonGUIApplication::EProcessPriority

Thread scheduling uses the following enum:

enum INonGUIThread::EThreadPriority

RELATED CONCEPTS

"Chapter 1. Open Class Threading Model" on page 1
"Behavior of IBM Open Class Threads" on page 2
"Resources and Conditions" on page 3
"Thread-Specific Data" on page 4

RELATED TASKS

"Start a Thread" on page 6

Multi-Processing Interface

Just as a user can interactively launch additional processes while one process is already running, a process itself can programmatically spawn secondary processes. IBM Open Class's process API is both object-oriented and open, allowing you to spawn external processes on Windows NT, OS/2, and AIX with the same source code.

These components constitute the process API:

Component	Description
IExternalProcess	Use to spawn a new process
INonGUIApplication	Use to control the external process
IApplication	Use to control a GUI process
ICurrentNonGUIApplication	Use to access the current active process
ICurrentApplication	Use to access the current active GUI process

Component	Description
IProcessId	Use to identify a process
enum EProcessPriority	Use to set the priority for running a process relative to other concurrent processes

Each process can spin multiple threads to run within it. Therefore, an Open Class application can be both multi-process and multi-threaded.

RELATED CONCEPTS

"Chapter 1. Open Class Threading Model" on page 1

RELATED TASKS

"Start a Thread"

Start a Thread

To start a thread within a process, use either INonGUIThread or IThread to represent the thread. Pass the code you want the thread to run to either the thread's constructor or to its start() method. You can wrap your thread code as a global function, as a class method, or as an object reference.

These examples pass a user-defined function, `mythreadedFunction()`, to a thread to run when it calls start(). Functions passed to thread constructors must be declared to be of type _Optlink or type _System (_Optlink and _System functions return void).

```
void _Optlink myThreadedFunction()
{
   // Your code, to be executed in
   // the new thread, goes here.
}
```

If you want to thread a function that has arguments, choose from the following types:

- _Optlink functions take a void* argument type
- _System functions take an unsigned long argument type

  ```
  void* threadArgs;
  ```

The following examples illustrate how to launch a thread to run your user-defined function:

- Pass the code you want threaded to the constructor of IThread or INonGUIThread; the new thread waits for the object's call to start() to begin execution.

  ```
  void main()
  {
     INonGUIThread myThread(myThreadedFunction, threadArgs);
     // ...
     myThread.start();
  }
  ```

- Call the default constructor for either IThread or INonGUIThread; pass the code that you want threaded to start().

```
void main()
{
   IThread *myThread = new IThread();
   // ...
   myThread->start(myThreadedFunction, threadArgs);
}
```

- Pass a pointer to an object of a class derived from IThreadFn, in which you have implemented run(), to either an IThread or INonGUIThread constructor.

  ```
  // Assume myThreadFn is a subclass of IThreadFn
  IThread myThread(myThreadFn,threadArgs);
  // ...
  myThread.start();
  // myThread.start() automatically executes
  // myThreadFn::run()
  ```

  You must create the IThreadFn* pointer with operator new; the IBM Open Class thread classes will delete this pointer after the thread has been started.
- Pass a pointer to an object of a class derived from IThreadFn, in which you have implemented run(), to either the start() function of an IThread or INonGUIThread object.

  ```
  IThread myThread();
  // ...
  // Assume myThreadFn is a subclass of IThreadFn
  myThread.start(myThreadFn,threadArgs);
  // myThread.start() automatically executes
  // myThreadFn::run()
  ```

  You must create the IThreadFn* pointer with operator new; the IBM Open Class thread classes will delete this pointer after the thread has been started.

The following example code demonstrates the use of the INonGUIThread class. It creates two classes derived from IThreadFn. Each of these classes overrides the IThreadFn::run() member function. The main function creates a thread for each of these classes, then starts both threads:

```
#include <ingthrd.hpp>
int done_1 = 0;
int done_2 = 0;
class TestFn : public IThreadFn
{
public:
   virtual void run()
   {
      cout << "First line, TestFn" << endl;
      INonGUIThread::current().sleep(10);
      cout << "Second line, TestFn" << endl;
      INonGUIThread::current().sleep(10);
      cout << "Third line, TestFn" << endl;
      INonGUIThread::current().sleep(10);
      cout << "Fourth line, TestFn" << endl;
      INonGUIThread::current().sleep(10);
      cout << "Fifth line, TestFn" << endl;
      done_1 = 1;
    }
};
class TestFn2 : public IThreadFn
{
public:
   virtual void run()
   {
      cout << "First line, TestFn2" << endl;
      cout << "Second line, TestFn2" << endl;
      cout << "Third line, TestFn2" << endl;
      cout << "Fourth line, TestFn2" << endl;
```

```
      cout << "Fifth line, TestFn2" << endl;
      done_2 = 1;
   }
};
void main()
{
   TestFn  *aThreadObj1 = new TestFn;
   TestFn2 *aThreadObj2 = new TestFn2;
   INonGUIThread myThread1;
   INonGUIThread myThread2;
   myThread1.start(aThreadObj1);
   myThread2.start(aThreadObj2);
   while (1)
   {
      if (done_1 && done_2)
         break;
   }
}
```

The following is the output of the above code:

```
First line, TestFn
First line, TestFn2
Second line, TestFn2
Third line, TestFn2
Fourth line, TestFn2
Fifth line, TestFn2
Second line, TestFn
Third line, TestFn
Fourth line, TestFn
Fifth line, TestFn
```

RELATED CONCEPTS

"Chapter 1. Open Class Threading Model" on page 1
"Behavior of IBM Open Class Threads" on page 2
"Resources and Conditions" on page 3
"Thread-Specific Data" on page 4
"Thread Scheduling" on page 5
"Multi-Processing Interface" on page 5

Chapter 2. Reference Counting

Reference counting is the technique of keeping track of how many pointers refer to a given object. This technique helps prevent memory errors. For example, suppose you have several pointers referring to a given object. If you delete all those pointers and forget to delete the object, your program will leak memory. Reference counting alleviates this by deleting an object automatically when no pointers refer to it.

A *reference counted object* is an object that keeps track of the number of pointers that refer to it. This number is called the *reference count*. When the reference count reaches zero, the reference counted object is automatically deleted. You create reference counted objects with the IMRefCounted class. This class has two member functions that change the reference count:

- addRef() increments the reference count by one
- removeRef() decrements the reference count by one

The IMRefCounted class is similar to the IRefCounted class provided by the IBM Open Class library in the past. This version of IMRefCounted differs in two ways:

- IMRefCounted is *thread safe*. That is, its addRef(), removeRef(), and count() member functions can be called from multiple threads simultaneously without causing data corruption.
- When an IMRefCounted object is created, its initial reference count is set to 0 rather than 1, which is what IRefCounted uses. An initial count of 0 makes IMRefCounted work much more cleanly with counted pointer classes such as ICountedPointerTo.

Counted pointers are objects that behave like C++ pointers that you use for reference counting. Use the ICountedPointerTo templatized class when creating counted pointers. You can use this class for reference counting both IMRefCounted objects and objects that do not descend from IMRefCounted.

The ICountedPointerTo class automatically calls addRef() and removeRef() on the object to which it points. In contrast, you have to remember to call addRef() and removeRef() yourself whenever you create and throw away a regular C++ pointer to a reference counted object. This is a large source of potential errors. If you forget to call removeRef(), your program will leak memory. If you forget to call addRef(), an object might be deleted while you are using it.

When you use ICountedPointerTo for reference counting objects that do *not* descend from IMRefCounted, you must follow some rules that are described in the task Use Counted Pointers.

RELATED TASKS

"Use Counted Pointers" on page 11
"Use Reference Counted Objects"

Use Reference Counted Objects

You cannot use the IMRefCounted class directly; you must derive a new class from it. To enforce this, its constructors and destructor are all declared protected.

When using an IMRefCounted subclass, you can call the following public functions:

Function	Description
addRef()	Increments the object's reference count by one. Use this method when you create a new pointer that aliases a reference counted object.
removeRef()	Decrements the object's reference count by one. Use this method when you are finished using a pointer that aliases the object.
count()	Return the object's current reference count. This is useful when you want to implement copy-on-write semantics, because it allows you to find out if anyone else is using the object.

When you are using IMRefCounted objects with raw C++ pointers, you must remember to call addRef and removeRef at the appropriate times, as in the following example:

```
class Foo : public IMRefCounted
{
    // ...
};
//...
    Foo *fooPtr = new Foo();
    // Added a pointer, so increase the ref count
    fooPtr->addRef();
    // Could increase the ref count if it needs to.
    functionCall(fooPtr);
    // Finished using fooPtr
    fooPtr->removeRef();
}
```

To implement copy-on-write semantics for a class, you typically use an internal implementation object that is reference counted and copied only when needed, as in the following example:

```
class Bar
{
   public:
      void setValue(int i);
   private:
      // BarImp is IMRefCounted
      BarImp *fImp;
};

void Bar::setValue(int i)
{
    if (fImp->count() > 1)
    {
         // My implementation is shared,
         // so make my own private copy
         BarImp *temp = new BarImp(*fImp);
         fImp->removeRef();
         fImp = temp;
         fImp->addRef();
    }
    fImp->assign(i);
}
```

Deriving a Class from IMRefCounted
When deriving your own class of IMRefCounted, the only real issue is whether you want to *force* clients to use your class in a reference-counted manner, or whether you want to allow them to use the class with normal stack or heap allocation semantics. If you want to force clients to do reference counting, make the destructor protected. This will prevent clients from allocating your class on the stack and from deleting heap objects of that class; they will instead have to call the removeRef() method that it inherits from IMRefCounted.

RELATED CONCEPTS

"Chapter 2. Reference Counting" on page 9

RELATED TASKS

"Use Counted Pointers"

Use Counted Pointers

The ICountedPointerTo is the counted pointer class provided by the IBM Open Class. The ICountedPointerTo class automatically calls addRef() and removeRef() on the object to which it points. In contrast, you have to remember to call addRef() and removeRef() yourself whenever you create and throw away a regular C++ pointer to a reference counted object. The following is an ICountedPointerTo equivalent to the first example in the task Use Reference Counted Objects:

```
class Foo : public IMRefCounted
{
   // ...
};
// ...
   ICountedPointerTo<Foo> fooPtr = new Foo(); // Calls addRef()
   functionCall(fooPtr);
   // removeRef called automatically by fooPtr's destructor()
```

Implementing copy-on-write semantics for a class is easier with the ICounterPointedTo class. The following example is equivalent to the second example in the task Use Reference Counted Objects:

```
class Bar
{
   public:
      void setValue(int i);
   private:
      // BarImp is IMRefCounted
      ICountedPointerTo<BarImp> fImp;
};
void Bar::setValue(int i)
{
   if (fImp->count() > 1)
   {
      // My implementation is shared,
      // so make my own private copy
      fImp = new BarImp(*fImp);
   }
   fImp->setValue(i);
}
```

The statement `fImp = new BarImp(*fImp);` works because ICountedPointerTo's assignment operator is smart enough to call removeRef and addRef on the old and new objects that are pointed to by the ICountedPointerTo object. For details, see the Assignment and Copying section below.

Use Reference Counting with Non-IMRefCounted Objects

Sometimes you want to use reference counting on objects of an existing class that does not inherit from IMRefCounted. ICountedPointerTo allows you to do this easily. ICountedPointerTo takes one of your existing classes as its template argument. If this class does not inherit from IMRefCounted, ICountedPointerTo adds a reference-counted wrapper around your objects to enable reference counting. For example, the following code will still work even though `Foo` does not inherit from IMRefCounted:

```
class Foo
{
   // ...
};
// ...
   // The ICountedPointerTo constructor will create a
   // reference-counted wrapper around the Foo object.
   ICountedPointerTo<Foo> fooPtr = new Foo();
   functionCall(fooPtr);
   // removeRef called automatically by fooPtr's destructor
```

In order for this to work, you must *always* use ICountedPointerTo rather than raw C++ pointers to ensure that there is only one wrapper object for each object you want to reference count. Because ICountedPointerTo automatically calls addRef and removeRef for you,

```
   ICountedPointerTo<Foo> fooPtr = new Foo();
   // Create a raw, "unwrapped" pointer...
   Foo *rawPtr = fooPtr.getAlias();
   // This line will create a second smart pointer with its own wrapper
   // around the Foo object.
   ICountedPointerTo<Foo> fooPtr2 = rawPtr;
   // This assignment statement will call removeRef on the original
   // wrapper, which will cause the Foo object to be deleted.
   fooPtr = 0;
   // Since the Foo object has been deleted, this will crash
   fooPtr2->function();
```

To do this correctly, simply omit the raw Foo* pointer and use ICountedPointerTo throughout your code:

```
   ICountedPointerTo<Foo> fooPtr = new Foo();
   // Create a second smart pointer that uses the same wrapper.
   ICountedPointerTo<Foo> fooPtr2 = fooPtr;
   // Calls removeRef on the wrapper, so its count will now be 1
   fooPtr = 0;
   // The Foo object has not been deleted yet, so this will work
   fooPtr2->function();
```

Assignment and Copying methods

All of the ICountedPointerTo constructors and assignment operators will perform reference counting correctly. The following table describes the constructors and assignment operators:

Constructor or Operator	Description
ICountedPointerTo (AType* adopt)	This constructor will "adopt" the object passed in and will increment its reference count. All further references to the object should be made through the ICountedPointerTo object; the raw pointer that was passed in to this constructor should be discarded.

Constructor or Operator	Description
ICountedPointerTo (const ICountedPointerTo<AType>& share)	The copy constructor will create another ICountedPointerTo object that aliases the same object. It will, of course, increment the object's reference count by 1.
operator = (AType* adopt)	Assigning a raw pointer into an ICountedPointerTo will "adopt" the object passed in and will increment its reference count by 1. All further references to the object should be made through the ICountedPointerTo object; the raw pointer that was passed in to this constructor should be discarded. If this ICountedPointerTo already pointed to an object before the assignment took place, the old object's reference count will decremented by 1 and the object will be deleted if the reference count reaches 0.
operator = (const ICountedPointerTo<AType>& share)	This assignment operator causes the counted pointer to refer to the same object as the ICountedPointerTo that is passed in. That object's reference count is incremented by 1. If this ICountedPointerTo already pointed to an object before the assignment took place, the old object's reference count will decremented by 1 and the object will be deleted if the reference count reaches zero. This operator handles self-assignment properly.
˜ICountedPointerTo()	If the counted pointer refers to an object, the destructor will decrement the object's reference count. If the reference count reaches zero, the object will be deleted.

Streaming Functions

ICountedPointerTo provides streaming operators that let you stream it to and from data streams. Writing an ICountedPointerTo to a stream does not write the pointer itself; it writes the object to which the ICountedPointerTo points. If you stream out several different counted pointers to the same object, the object will only be streamed out the first time, and aliases to it will be streamed out after that. For example:

```
IDataStream *stream = ....;
ICountedPointerTo<Foo> foo1 = new Foo();
ICountedPointerTo<Foo> foo2 = foo1;
foo1 >>= *stream;    // Streams out the Foo object
foo2 >>= *stream;    // Just streams out an alias
// ...
ICountedPointerTo<Foo> foo3, foo4;
foo3 <<= *stream;    // Streams in the Foo object
foo4 <<= *stream;    // Creates an alias to the same Foo object
// foo3 and foo4 now point to the same Foo object.
```

These streaming methods will only work if the object being pointed to inherits from IMStreamable.

The following example demonstrates how to create and use a counted pointer, as well as how to stream objects pointed to by counted pointers in and out of data streams:

```
// Counted pointer example
#include <icntptr.hpp>
#include <istrmmod.hpp>
// IStreamModules are used with IDataStreams. They provide
// a name-space like facility to help avoid conflicts
// between the names of classes that may be written to a
// stream.
IStreamModule myModule("myName");
class MyDatabase : public virtual IMStreamable, public IMRefCounted
{
   // All non-abstract classes deriving from IMStreamable
   // must include the StreamableDeclarations and
   // StreamableDefinitions macros.
   StreamableDeclarationsMacro(MyDatabase);
   public:
      float getName(void)
         { return name; }
      void setName(float data)
         { name = data; }
   protected:
     virtual void writeToStream(IDataStream&) const;
     virtual void readFromStream(IDataStream&);
   private:
      float name;
};
// All subclasses of IMStreamable must override
// the writeToStream() and readFromStream()
// functions.
void MyDatabase::writeToStream(IDataStream& toWhere) const
{
   name >>= toWhere;
}
void MyDatabase::readFromStream(IDataStream& fromWhere)
{
  name <<= fromWhere;
}
// All non-abstract classes deriving from IMStreamable
// must include the StreamableDeclarations and
// StreamableDefinitions macros.
StreamableDefinitionsMacro(MyDatabase, myModule);
int main(void)
{
   cout << "Create first counted pointer..." << endl;
   ICountedPointerTo<MyDatabase> firstUser = new MyDatabase();
   cout << "The count of the first counted pointer: "
        << firstUser->count() << endl << endl;
   cout << "Create two more counted pointers..." << endl;
   ICountedPointerTo<MyDatabase> secondUser(firstUser);
   ICountedPointerTo<MyDatabase> thirdUser = secondUser;
   cout << "The count of the first counted pointer: "
        << firstUser->count() << endl << endl;
   cout << "Put data into object pointed to by the first counted pointer..." << endl;
   firstUser->setName(123456);
   cout << "Data in object pointed to by second counted pointer: "
        << secondUser->getName() << endl << endl;
   cout << "Remove third counted pointer..." << endl;
   thirdUser = 0;
   cout << "The count of the first counted pointer: "
        << firstUser->count() << endl << endl;
   // The remaining code demonstrates how to stream
   // ICountedPointerTo ojects in and out of data
   // streams.
   cout << "Create memory stream..." << endl;
   IDataStream *stream = IDataStream::createMemoryStream();
   cout << "Stream out the MyDatabase object..." << endl;
   firstUser >>= *stream;
   cout << "Reset the stream..." << endl;
   stream->reset();
   cout << "Stream in the MyDatabase object into a fourth counted pointer..." << endl;
   ICountedPointerTo<MyDatabase> fourthUser;
   fourthUser <<= *stream;
   cout << "Data in object pointed to by fourth counted pointer: "
        << fourthUser->getName() << endl << endl;
   cout << "The count of the first counted pointer: "
        << firstUser->count() << endl << endl;
   cout << "The count of the fourth counted pointer: "
        << fourthUser->count() << endl << endl;
}
```

Output
The following is the output of the above example:

```
Create first counted pointer...
The count of the first counted pointer: 1
Create two more counted pointers...
The count of the first counted pointer: 3
Put data into object pointed to by the first counted pointer...
Data in object pointed to by second counted pointer: 123456
Remove third counted pointer...
The count of the first counted pointer: 2
Create memory stream...
Stream out the MyDatabase object...
Reset the stream...
Stream in the MyDatabase object into a fourth counted pointer...
Data in object pointed to by fourth counted pointer: 123456
The count of the first counted pointer: 2
The count of the fourth counted pointer: 1
```

The above example creates three counted pointers, `firstUser`, `secondUser`, and `thirdUser`, that point to the same `MyDatabase` object. The example then streams out the object from `firstUser` and streams in that object to `fourthUser`. Note that `firstUser` and `fourthUser` do not point to the same object.

Other Functions
ICountedPointerTo also provides the usual comparison operators, plus a number of other useful functions:

Function	Description
getAlias()	Returns the underlying object to which this ICountedPointerTo points. If it points to nothing, 0 will be returned.
valid()	Returns true if this ICountedPointerTo refers to an object and false if it does not.
validate()	This method is essentially an assertion that the ICountedPointerTo is valid. If it is not, an IInvalidRequest exception will be thrown with its error code set to IC_NULL_IREFERENCE.
count()	Returns the current reference count of the object to which this pointer refers. If there is no object, it returns 0.

Function	Description
copyPointer()	This global function can be used to do a polymorphic copy of an IMStreamable object that is pointed to by an ICountedPointerTo. If the object's class does not inherit from IMStreamable, you will get a compile-time error. If there is no object being pointed to, this function will return 0.

RELATED CONCEPTS

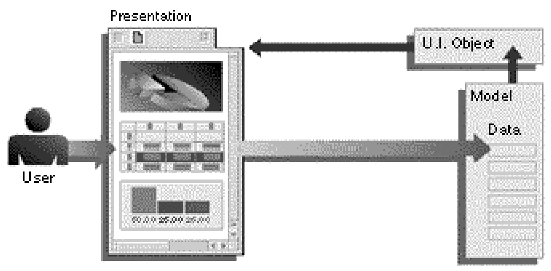
"Chapter 2. Reference Counting" on page 9

RELATED TASKS

"Use Reference Counted Objects" on page 9

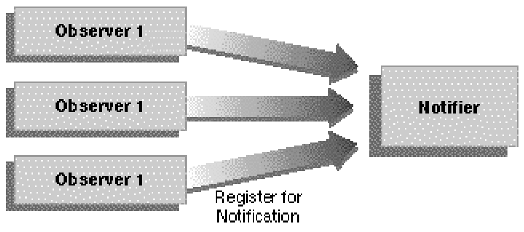
Chapter 3. Event Notification

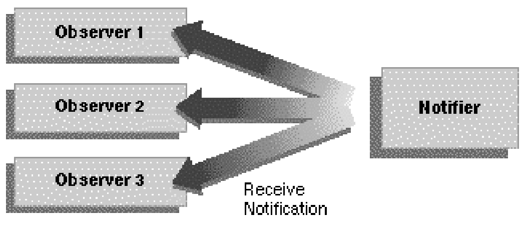
You use the Notification classes to enable an object to be notified by other objects of any changes which it needs to know about. For example, in a compound document application which uses the model/view/presenter design, whenever the state of the data in a model changes, any views opened on the model that the change affects need to be informed so that the views can present the change to the user.



Your model should *notify* the view when the model's data is modified so that the view can redraw itself to reflect the change.

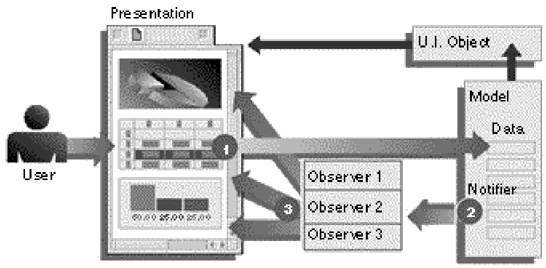
An object which issues notification to other objects of changes in its state is referred to as a *notifier*. The objects which receive such change notifications are referred to as *observers*. To become an observer, an object must *register* with the object whose state it wishes to observe. (To register an observer, call its IObserver::handleNotificationsFor member function.) Once registered, when that object is modified, it issues notification to the registered observer objects, passing a *notification event*.





Before you delete an observer object you must *unregister* with the notifier. (To unregister an observer, call its IObserver::stopHandlingNotificationsFor member function.)

Any application object can use a notifier to notify observers of particular events or changes in its state.



In this example, the user interface can update the display whenever the model completes a recalculation requested by the user.

For other examples, a stock market ticker-tape component that constantly monitors stock reports needs to inform its associated view of changes to the incoming data, while a clock component needs to keep its display constantly updated.

Notification Solutions

The notification classes implement three generic solutions:

- Synchronizing the states of two related objects
- Filtering notifications in an application where one source can generate many events, such as might happen with a feature-rich user interface window
- Using asynchronous notification to eliminate one event's blocking another, such as a print or save request preventing further data entry

IBM Open Class extends its notification services in two important ways:

- It offers asynchronous, in addition to synchronous, notification
- It provides a filtering mechanism so that the client object can specify what types of notification to receive

These enhancements lower the risk of application deadlocks, provide a more flexible programming model, and facilitate scalability. They can be used for either single-threaded or multithreaded applications.

RELATED CONCEPTS

"Notification Classes" on page 19"Notification Structure" on page 20

RELATED TASKS

"Notify Observers Synchronously" on page 21
"Notify Observers Asynchronously" on page 25

Notification Classes

When an event requiring notification occurs, the notifier constructs an INotificationEvent object defining the type of event and sends it to any observers registered for that type of event. Each observer is responsible for handling the event as appropriate.

The following class objects establish a notification pattern between application objects:

- *Notifiers* provide the protocol needed by a notifier object to issue change messages to observer objects.
- *Observers* define the protocol used by observers to connect to, and receive notifications from a notifier object.
- *Interests* are used by observers to identify the specific event for which they wish notification.
- *Notification Events* identify the notification type and are passed from the notifier to one or more observers.

Notifiers
The notifier classes provide two classes that support the creation of objects capable of communicating changes in their state to other objects.

- INotifier is an abstract base class that defines the notifier.
- IStandardNotifier is a derived class that concretely implements the notifier.

Notifier objects are responsible for defining the list of their supported notification events, for managing lists of observers, and for notifying registered observers when an event of interest occurs.

Observers
Observer classes define the protocol used by observers to register for and receive notifications. The abstract base class IObserver is templatized into two versions: IObserverConnectionTo, which takes only the target notifier; and IObserverForConnectionTo, which takes both the target notifier and a specific event notification data type.

Interests
An observer uses the IInterest class to indicate the types of notification it wants to receive. IInterest provides helper functions that notifiers use to notify observers of the events they are interested in. A notifier uses IInterest and INotificationEventFor to pass event data to observers.

You can streamline your notification mechanisms by filtering the types of notification an observer receives. The notifier contains a set of notification IDs that defines all the possible types of changes, and observers can register to be notified only of the types of changes they care about. The following is the typedef provided for notification IDs:

```
typedef const char *INotificationId;
```

You use INotificationId to create simple strings that uniquely describe a particular type of change that might originate notifications. Interests and notification events use this ID to identify the type.

Notification Event Types
INotificationEvent encapsulates a single notification event. Notification event

objects identify the notification type, and are passed from the notifier to the observers. INotificationEventFor is a templatized class used for passing event data from a notifier to its observers.

RELATED CONCEPTS

"Chapter 3. Event Notification" on page 17
"Notification Structure"

RELATED TASKS

"Notify Observers Synchronously" on page 21
"Notify Observers Asynchronously" on page 25

Notification Structure

Notification Implementation
When an event requiring notification occurs, the notifier constructs an INotificationEvent object defining the type of event and sends it to any observers registered for that type of event. Each observer is responsible for handling the event as appropriate.

To implement your own notification mechanism your code must follow a basic order, or sequence, of calls:

1. Create a notifier.
2. Create an observer.
   a. Connect the observer to the notifier.
   b. Register with the notifier with an interest for a change or event.
3. Use your notifier from the application object managing the data or event.

Asynchronous or Synchronous Notification
You can send notifications synchronously or asynchronously, depending on the requirements of both the notifier and the observer. Some notifiers require that notifications be handled synchronously so that all notification processing is complete before returning to the notifier. A system shutdown service is an example of this.

Other notifiers, such as a file server, might require that notifications be delivered asynchronously so that they are not blocked for long periods of time, waiting for observers to complete notification handling.

The Notification Framework provides two INotifier methods that you can use depending on your needs. INotifier::notifyObservers() provides a synchronous implementation and INotifier::notifyObserversAsync() provides one that is asynchronous.

When you call INotifier::notifyObservers(), the notifier calls the notify member function of all the connections interested in that notification in the same thread in which INotifier::notifyObservers() was called. When the call completes, the notifier is assured that all observers interested in the notification have received it.

However, this means that an observer can tie up the notifier by taking a long time to process a notification.

When you call INotifier::notifyObserversAsync(), a request to perform the notification is passed to a request processor. The notifier can then immediately continue to perform other work. When the request processor eventually delivers the notification, receivers cannot make any assumptions about the state of the notifier (even that it still exists).

Multi-threaded Notification
To register an observer, an observer's handleNotificationFor() function is called. It, in turn, chains into the notifier's addObserver() function which can then record the threadID associated with the observer.

When an event for which the observer is registered occurs, in the case of synchronous multi-threaded notification, the notifier's notifyObservers() function is called.

The notification is dispatched on the same thread on which the observer was created. This is basically the same behavior as if notifyObserversAsync() were invoked instead of notifyObservers().

For asynchronous multi-threaded notification, when the notifier's notifyObserversAsync() function is called, it determines whether it needs to send a notification to an observer in another thread. If so, it posts to the request queue associated with the observer's thread.

When using asynchronous notification, it is important to start the notification dispatcher properly:

- For a GUI thread, one that uses IThread::current().processMsgs(), your main function should have a processMsgs() loop to handle the GUI events for the thread.
- If you create observers in a non-GUI thread, one that does not use IThread::current().processMsgs(), and you want the observers to receive asynchronous notifications, then you should use ICrossThreadNotificationLoop() in the non-GUI thread to dispatch to asynchronous notification-aware observers. You would include code similar to the following:

```
ICrossThreadNotificationLoop theLoop(anObserver);
theLoop.run();
```

RELATED CONCEPTS

"Chapter 3. Event Notification" on page 17
"Notification Classes" on page 19

RELATED TASKS

"Notify Observers Synchronously"
"Notify Observers Asynchronously" on page 25

Notify Observers Synchronously

Notification requires a notifier and at least one observer. Your code must instantiate these objects:

1. A notifier derived from IStandardNotifier
2. An observer derived from IObserver

In the notifier, perform the following steps:

1. Define your *interests*. Interests are data members of type IInterest that you want to observe.
2. Define your *getters*. Getters are member functions that serve as an interface for your interests. Each getter should return a reference to one of your interests.
3. Define your *setters*. Setters are the member functions that will notify the observers by sending them the appropriate interest. In each setter, call notifyObservers(). The argument for notifyObervers() should be an INotificationEvent object instantiated with the corresponding getter.
4. In the notifier's constructor, call enableNotification().

In the observer, for each interest you wish to observe, perform the following steps:

1. Create a handler for each interest that you wish to observe. These handlers take an INotificationEvent object as an argument.
2. For each handle, create an IObserverConnectionTo object. These objects are your *connections*.
3. In the constructor for the observer, register each connection with the handleNotificationsFor() member function. Use the corresponding getter for the argument.
4. In the destructor, be sure to unregister the connection with the stopHandlingNotificationsFor() function.

The following sample demonstrates the use of interest-based, synchronous notification.

```
// Synchronous notification example
#include <istdntfy.hpp>
#include <iobservr.hpp>
#include <inotifev.hpp>
class MyCustomer : public IStandardNotifier
{
public:
   // The constructor creates the three supported
   // interests and enables notification. Any
   // observer interested in a name change, address
   // change, or home phone change will register with
   // these interests.
   MyCustomer() :
      fNameChangedInterest     (*this, nameId),
      fAddressChangedInterest  (*this, addressId),
      fHomePhoneChangedInterest(*this, homePhoneId)
      {
         enableNotification();
      }
      virtual ~MyCustomer() {};
      virtual IString name() {return fName;}
      virtual IString address() {return fAddress;}
      virtual IString homePhone() {return fHomePhone;}
      // These are the setters for MyCustomer. The
      // setters will make the requested change and then
      // notify observers by sending the appropriate
      // interest via an INotificationEvent.
      virtual void setName(IString newname)
      {
         fName = newname;
         notifyObservers
            (INotificationEvent(nameChangedInterest()));
      }
      virtual void setAddress (IString newaddress)
      {
         fAddress = newaddress;
         notifyObservers
            (INotificationEvent(addressChangedInterest()));
      }
      virtual void setHomePhone(IString newphone)
      {
         fHomePhone = newphone;
         notifyObservers
            (INotificationEvent(homePhoneChangedInterest()));
      }
      // These are the getters for the three supported
      // interests. Any observer interested in a name
      // change, address change, or home phone change
      // will register with these interests.
      IInterest& nameChangedInterest ()
      {
         return fNameChangedInterest;
      }
      IInterest& addressChangedInterest ()
      {
         return fAddressChangedInterest;
      }
      IInterest& homePhoneChangedInterest ()
      {
         return fHomePhoneChangedInterest;
      }
private:
   // Our data members
   IString fName;
   IString fAddress;
   IString fHomePhone;
   // These are interest members corresponding to the
   // observable data members
   IInterest fNameChangedInterest;
   IInterest fAddressChangedInterest;
   IInterest fHomePhoneChangedInterest;
   // Declare the NotificationIds
   static INotificationId const
      nameId,
      addressId,
      homePhoneId;
};
// Define the INotificationIds
const INotificationId MyCustomer::nameId
   = "MyCustomer::name";
const INotificationId MyCustomer::addressId
   = "MyCustomer::address";
const INotificationId MyCustomer::homePhoneId
   = "MyCustomer::homePhone";

class MyObserver
{
public:
   // constructor for observer
   MyObserver (MyCustomer& aCustomer);
   virtual ~MyObserver();
   // handler method to handle any change in the customer
   virtual void
      handleAnyChange(const INotificationEvent& anEvent)
   {
      printf(">> Why hello, MyObserver::handleAnyChange ");
      printf("just received the event: %s \n",
             anEvent.notificationId());
   }
   // handler method to handle just name changes
   // in the customer
   virtual void
      handleNameChange(const INotificationEvent& anEvent)
   {
      printf(">> Why hello, MyObserver::handleNameChange ");
      printf("just received the event: %s \n",
              anEvent.notificationId());
   }
private:
   // for each event you are interested in, create
   // a connection object
   IObserverConnectionTo<MyObserver>
      fAnyChangeConnection;
   IObserverConnectionTo<MyObserver>
      fNameChangeConnection;
   MyCustomer& fCustomer;
};

// connect this observer to the given customer notifier

MyObserver::MyObserver(MyCustomer& aCustomer) :
   fCustomer(aCustomer),
   // all events trigger this method
   fAnyChangeConnection
      (*this, MyObserver::handleAnyChange),
   // only name change events trigger this method
   fNameChangeConnection
      (*this, MyObserver::handleNameChange)
{
   // this connection handles all notifications
   // from aNotifier
   fAnyChangeConnection.handleNotificationsFor(fCustomer);
   // this connection handles only name change
   // notifications from aNotifier
   fNameChangeConnection.handleNotificationsFor
      (fCustomer.nameChangedInterest());
}
// unregister from receiving events from the customer
MyObserver::~MyObserver()
{
   fAnyChangeConnection.stopHandlingNotificationsFor(fCustomer);
   fNameChangeConnection.stopHandlingNotificationsFor(fCustomer);
}

void main()
{
   MyCustomer theCustomer;
   MyObserver theObserver(theCustomer);

   // should trigger handleAnyChange/handleNameChange
   theCustomer.setName("Raymond the IOC Writer");

   // should only trigger handleAnyChange
   theCustomer.setAddress("1150 Eglington Ave.");
}
```

The class MyCustomer is an example of deriving from an IStandardNotifier. The class MyCustomer encapsulates a name, address, and home phone number. It supports getting and setting its fields and notifies any registered observers when the name, address, or home phone number changes.

The MyObserver class receives notifications. It uses IObserverConnections to direct notifications to specific methods. In this example, it receives notifications from a customer object.

Here is the output from the program:

```
>> Why hello, MyObserver::handleAnyChange just received the event:
   MyCustomer::name
>> Why hello, MyObserver::handleNameChange just received the event:
   MyCustomer::name
>> Why hello, MyObserver::handleAnyChange just received the event:
   MyCustomer::address
```

RELATED CONCEPTS

"Chapter 3. Event Notification" on page 17
"Notification Classes" on page 19
"Notification Structure" on page 20

RELATED TASKS

"Notify Observers Asynchronously"
"Pass Data Along with Event Notification" on page 29

Notify Observers Asynchronously

To notify asynchronously, follow the same steps as for synchronous notification, substituting a call to notifyObserversAsync for notifyObservers. You can implement truly asynchronous notification by putting the notifier and the observer in different threads.

The following procedures show you how to notify asynchronously by putting each observer on a non-GUI thread:

- Asynchronous notification requires a notifier and at least one observer. Instantiate the following objects in your code:
    1. A notifier derived from IStandardNotifier
    2. For each observer, a class derived from IObserver that inherits publicly from IThreadFn
- In the notifier, perform the following steps:
    1. Define your *interests*. Interests are data members of type IInterest that you want to observe.
    2. Define your *getters*. Getters are member functions that serve as an interface for your interests. Each getter should return a reference to one of your interests.
    3. Define your *setters*. Setters are the member functions that will notify the observers by sending them the appropriate interest. In each setter, call notifyObserversAsync. The argument for notifyOberversAsync should be an INotificationEvent object instantiated with the corresponding getter.
    4. In the notifier's constructor, call enableNotification().
- In each observer, perform the following steps:
    1. Create a *handler* the interest the observer will observe. This handler is a member function that takes an INotificationEvent object as an argument.
    2. Override the IThreadFn::run() member function as follows:
        a. Create an IObserverConnectionTo object from the interest you want to observe. This object is a *connection*.
        b. Register the connection with the handleNotificationsFor() member function. Use the corresponding getter for the argument.
        c. Create an ICrossThreadNotificationLoop object from the connection.
        d. Call the ICrossThreadNotificationLoop::run() member function.
- In the main function, perform the following steps:
    1. Create your observers.
    2. Create an INonGUIThread object for each observer.
    3. Start each thread with a reference to its corresponding observer.

The following example code demonstrates the use of interest-based, asynchronous notification. The output of this program is the same as the example code presented in the "Notify Synchronously" task:

```
// Notify Asynchronously
#include <istdntfy.hpp>
#include <iobservr.hpp>
#include <inotifev.hpp>
#include <ithread.hpp>
// Define the INotificationIds
const INotificationId MyCustomer::nameId
   = "MyCustomer::name";
const INotificationId MyCustomer::addressId
   = "MyCustomer::address";
const INotificationId MyCustomer::homePhoneId
   = "MyCustomer::homePhone";
class MyCustomer : public IStandardNotifier
{
public:
   // The constructor creates the three supported
   // interests and enables notification. Any
   // observer interested in a name change, address
   // change, or home phone change will register with
   // these interests.
   MyCustomer();
   virtual ~MyCustomer() {};
   virtual IString name() {return fName;}
   virtual IString address() {return fAddress;}
   virtual IString homePhone() {return fHomePhone;}
      // These are the setters for MyCustomer. The
      // setters will make the requested change and then
      // notify observers by sending the appropriate
      // interest via an INotificationEvent.
      virtual void setName(IString newname)
      {
         fName = newname;
         notifyObserversAsync
            (INotificationEvent(nameChangedInterest()));
      }
      virtual void setAddress (IString newaddress)
      {
         fAddress = newaddress;
         notifyObserversAsync
            (INotificationEvent(addressChangedInterest()));
      }
      virtual void setHomePhone(IString newphone)
      {
         fHomePhone = newphone;
         notifyObserversAsync
            (INotificationEvent
               (homePhoneChangedInterest()));
      }
      // These are the getters for the three supported
      // interest. Any observer interested in a name
      // change, address change, or home phone change
      // will register with these interests.
      IInterest& nameChangedInterest ()
      {
         return fNameChangedInterest;
      }
      IInterest& addressChangedInterest ()
      {
         return fAddressChangedInterest;
      }
      IInterest& homePhoneChangedInterest ()
      {
         return fHomePhoneChangedInterest;
      }
private:
   // Our data members
   IString fName;
   IString fAddress;
   IString fHomePhone;
   // These are interest members corresponding to the
   // observable data members
   IInterest fNameChangedInterest;
   IInterest fAddressChangedInterest;
   IInterest fHomePhoneChangedInterest;
   // Declare the NotificationIds
   static INotificationId const
      nameId,
      addressId,
      homePhoneId;
};
class MyNameChangeObserver: public IThreadFn
{
   public:
      // Constructor:
      MyNameChangeObserver::MyNameChangeObserver
         (MyCustomer& aCustomer): fCustomer(aCustomer)
      {}
      // handler function to handle any change in the customer
      virtual void
         handleNameChange(const INotificationEvent& anEvent)
      {
         printf(">> Why hello, myObserver::handleNameChange ");
         printf("just received the event:\n   %s \n",
              anEvent.notificationId());
      }
      virtual void run();
   private:
      // for each event you are interested in, create
      // a connection object
      MyCustomer& fCustomer;
};
class MyAnyChangeObserver : public IThreadFn
{
public:
   // constructor for observer
   MyAnyChangeObserver::MyAnyChangeObserver(MyCustomer& aCustomer) :
      fCustomer(aCustomer)
   {}
   // handler method to handle any change in the customer
   virtual void
      handleAnyChange(const INotificationEvent& anEvent)
   {
      printf(">> Why hello, myObserver::handleAnyChange ");
      printf("just received the event:\n   %s \n",
             anEvent.notificationId());
   }
   // handler method to handle just name changes
   // in the customer
   virtual void run();
private:
   // for each event you are interested in, create
   // a connection object
   MyCustomer& fCustomer;
};
// MyCustomer constructor:
MyCustomer::MyCustomer()
   : fNameChangedInterest     (*this, nameId)
   , fAddressChangedInterest  (*this, addressId)
   , fHomePhoneChangedInterest(*this, homePhoneId)
{
   enableNotification();
}
// Overriding IThreadFn::run() defines the
// code to be executed in a thread
void MyNameChangeObserver::run()
{
   // for each event you are interested in, create
   // a connection object
   IObserverConnectionTo<MyNameChangeObserver>
      fNameChangeConnection
         (*this, &MyNameChangeObserver::handleNameChange);
   // this connection handles only name change
   // notifications from aNotifier
   fNameChangeConnection.handleNotificationsFor
      (fCustomer.nameChangedInterest());
   // Start the notification dispatcher
   ICrossThreadNotificationLoop
      loopNameChange(fNameChangeConnection);
   loopNameChange.run();
}
// Overriding IThreadFn::run() defines the
// code to be executed in a thread
void MyAnyChangeObserver::run()
{
   // for each event you are interested in, create
   // a connection object
   IObserverConnectionTo<MyAnyChangeObserver>
      fAnyChangeConnection
         (*this, &MyAnyChangeObserver::handleAnyChange);
   // this connection handles only name change
   // notifications from aNotifier
   fAnyChangeConnection.handleNotificationsFor
      (fCustomer);
   // Start the notification dispatcher
   ICrossThreadNotificationLoop
      loopAnyChange(fAnyChangeConnection);
   loopAnyChange.run();
}
void main()
{
   MyCustomer theCustomer;
   MyNameChangeObserver *theNameChangeObserver =
      new MyNameChangeObserver(theCustomer);
   MyAnyChangeObserver *theAnyChangeObserver =
      new MyAnyChangeObserver(theCustomer);
   INonGUIThread threadAnyChange;
   INonGUIThread threadNameChange;
   threadAnyChange.start(theNameChangeObserver);
   threadNameChange.start(theAnyChangeObserver);
   // should trigger handleAnyChange/handleNameChange
   INonGUIThread::current().sleep(1000);
   theCustomer.setName("Chester the Dog");
   // should only trigger handleAnyChange
   INonGUIThread::current().sleep(1000);
   theCustomer.setAddress("Colchester, Ontario");
}
```

RELATED CONCEPTS

"Chapter 3. Event Notification" on page 17
"Notification Classes" on page 19
"Notification Structure" on page 20

RELATED TASKS

"Notify Observers Synchronously" on page 21

Pass Data Along with Event Notification

To pass event data along with a notification of the event, follow the same steps as described in Notify Synchronously with these exceptions:

1. In your notifier, pass an INotificationEventFor object to notifyObservers instead of an INotificationEvent object. That is, use this:

   ```
   notifyObservers(INotificationEventFor
                     (myAlertInterestFunction(),
                        data));
   // data is of type MyEventDataClass
   ```

   instead of:

   ```
   notifyObservers(INotificationEvent
                     (myAlertInterestFunction());
   ```

   The constructor for an INotificationEventFor object takes two arguments: your interest, and the event data you want to pass along with the notification of your event.
2. In your observer, instead of creating connection objects of type IObserverConnectionTo, use the connection template class IObserverForConnectionTo. That is, use this:

   ```
   IObserverForConnectionTo<MyEventDataClass, MyObserver>
      myDataChangeConnection;
   ```

   instead of:

   ```
   IObserverConnectionTo<MyObserver>
      myDataChangeConnection;
   ```

   Notice that the constructor for IObserverForConnectionTo takes two data types in its constructor: the type of the event data that you want to pass, and the type of your observer.
3. The observer methods that are registered to handle the notification events have a INotificationEventFor object parameter corresponding to the one used by the notifier's notifyObservers method. Use this form of handleNotification():

   ```
   void handleNotification
      (const INotificationEventFor<MyEventDataClass>& event);
   ```

   instead of:

   ```
   void handleNotification(const INotificationEvent& event);
   ```
4. INotificationEventFor<MyEventDataClass> has the following member function:

   ```
   const MyEventDataClass& eventData() const;
   ```

   that provides a mechanism to get the event data back from the notification event.

RELATED CONCEPTS

"Chapter 3. Event Notification" on page 17
"Notification Classes" on page 19
"Notification Structure" on page 20

RELATED TASKS

"Notify Observers Synchronously" on page 21

Chapter 4. Application Resources

A *resource library* is the file, such as an executable, dynamic link library (Windows and OS/2) or shared library (AIX), with a compiled resource file bound to it.

The IDLLModule class supports the loading, unloading, and addressing of entry points in a dynamic link library (DLL) or shared library. These classes make it easier and safer to track loaded DLLs. It provides access to module information for those loaded DLLs, such as entry point addresses and the file name of the module.

Search for Resource Files
You may specify how your application searches for resources libraries with the IDynamicLinkLibrary of IDLLModule constructor. The enumerated type ESearchLocation defines the following search methods:

Search Method	Description
kOSDefaultSearch	Finds a resource library using the PATH or LIBPATH environment variables following the search rules of the native operating system. This is the default search method.
kNLSPathSearch	Finds the dynamic link library using the NLSPATH environment variable after substituting %L with the users' current locale. • WIN OS/2 On these platforms, it is the LANG environment variable. • AIX Instead of using the LANG environment, the user's current locale is retrieved from a call to setlocale(LC_MESSAGES, NULL). Therefore, IBM Open Class applications should call setlocale() properly if they want to load their libraries (and their message catalogues) from the NLSPATH Loading a resource library from NLSPATH allows you to have multiple languages of your resource libraries on a single system.

AIX

When loading a shared library using IDLLModule these classes will search paths specified in the header section of the executable in addition to the NLSPATH or the LIBPATH environment variables.