VisualAge C++ Professional for AIX



IBM Open Class Concepts

Version 4.0

VisualAge C++ Professional for AIX



IBM Open Class Concepts

Version 4.0

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

First Edition (June 1998)

This edition applies to Version 4.0 of the VisualAge C++ product, 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. All rights reserved.
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xi
Programming Interface Information xi
Trademarks and Service Marks xi
Industry Standards xii

About this Book xiii

Chapter 1. Introduction 1
Introduction to the Open Class Library 1
Frameworks and Classes. 1
Changes in this Release of Open Class 3
AIX Changes 3
Toolbar and Related Changes 3
IWindow Changes 5
Summary 13
Graphics 16
Boolean Type Definition 17
Scroll Bar 17
ICUSTOMBUTTON 17
IMOUSEPOINTERHANDLER 18
New Classes 18
New Interface in IWindow for Event Dispatching & Associated Changes to IMouseHandler. 18
Deprecated Functions 22
Standard Library Overview 26

Chapter 2. Collection Classes 29
Introduction to the Collection Classes 29
Types of Classes. 29
Hierarchy and Design of the Collection Classes 29
Collection Class Hierarchy 29
Overall Implementation Structure 30
The Based-On Concept 31
Element Functions and Key-Type Functions 32
Collection Class Polymorphism 32
Collection Characteristics. 32
Ordering of Collection Elements 33
Access by Key 33
Equality Relation 34
Uniqueness of Entries 34
Iteration 34
Iteration with Cursors 35
Iteration with allElementsDo. 36
Flat Collections 36
Combinations of Properties for Flat Collections. 36
Trees 37
Backward-Compatible Items. 38
Reference Classes 38
IIterator and IConstantIterator 38
The forCursor macro 38
IECOps 38
Naming Conventions 38
Incompatibilities 39

© Copyright IBM Corp. 1998

New class hierarchy . . . 39
The newCursor method . . . 39
Deriving from Reference Classes . . . 39
Collection Class Library Exceptions . . . 39
Exceptions Defined and Used by the Collection Classes . . . 39
Exception Causes . . . 40
Precondition-Violations . . . 40
Other Exceptions . . . 41
Exceptions Caused by System Failures and Restrictions . . . 41
Support for Notifications . . . 42
Thread Safety and the Collection Classes . . . 42
Restrictions . . . 43
Smart Pointers . . . 43
Smart Element Pointers . . . 43
Automatic Storage Management . . . 43
Generally Useful . . . 44

Chapter 3. Implementation Variants . . . 45
AVL Tree . . . 45
B* Tree . . . 46
Diluted Table . . . 47
Hash Table . . . 47
List . . . 49
Table . . . 49

Chapter 4. International Framework . . . 51
International Text and Applications . . . 51
Locales . . . 51
Collation . . . 51
Transcoding . . . 51
Locales in the Internationalization Framework . . . 51
Object Portability . . . 52
Names of International Objects . . . 52
Locale Classes . . . 52
Access to Locale Resources . . . 53
Locale Iteration . . . 54
Collation Classes . . . 55
Overview of Collation Classes . . . 55
Collation Subclasses . . . 56
Collation Iteration . . . 57
Ordering Strength . . . 57
Transcoding Classes . . . 58
Overview of Transcoding Classes . . . 58
Transcoders . . . 59
Line-Breaking Conversion . . . 60
Character Set Iteration . . . 61
Special Characters . . . 62
Exception Characters . . . 62
Mapping Proximity . . . 63

Chapter 5. Date and Time Classes . . . 65
Dates and Calendars . . . 65
Date and Time Classes . . . 65
Time . . . 65
Time Stamps . . . 66

Chapter 6. National Language Support and Double-Byte Character Sets . . 67
Overview of National Language Support 67
Double-Byte Character Set Support 67
DBCS and National Language Support. 67

Chapter 7. Text Framework 69
Text Overview 69
Text Creation and Manipulation 69
Text Boundaries 70
Identifying Text Boundaries 70
Character Boundaries 70
Word Boundaries. 70
Line Boundaries 70
Sentence Boundaries 71
Text Storage 71
Storage Mechanism. 71
Strings and Buffers 72
Overview of Strings and Buffers 72
String Buffers 73
String Classes. 73
String Comparison 74
String Indexes. 75
String Formats. 75
String Size and Capacity 76
IText versus IString 76
Text Display 77
Unicode Text 77
Static Text 77
Summary of Text Display Classes 79

Chapter 8. Styles 81
Styles and Style Sets 81
Style Classes 81
Character Style Classes 83
Paragraph Style Classes 84
Style Propagation 85
Character Data 86
Editing Character Data. 87
Text and Style Run Iteration. 87
Character Iteration 87
Style Run Iteration 88

Chapter 9. Character and Text Queries 91
Overview of Unicode Support 91
The IUnicode Class 91
Character Support 91
Supported Scripts 92
Reserved Areas 92
Character Values. 92
Character Name Enumerations 93
Character Properties 95

Chapter 10. Streams and Storage 97
Introduction to the File System. 97
Overview of File System Entities 97
Files 98

Directories . . . 98
Volumes . . . 98
Attributes. . . . 99
Overview of Pathnames and Pathname Parsers . . . 99
Host-Specific Pathnames . . . 100
Pathname Parsers . . . 100
Portable Pathnames . . . 100
Data Accessors . . . 100
File System Iterators . . . 102
IFileSystemEntity's moveTo and copyTo Functions . . . 102
Mover and Copier Classes . . . 102
Thread Safety . . . 103
File System Exceptions . . . 103

Chapter 11. Streaming Classes . . . 105
Introduction to the Streaming Classes . . . 105
Exceptions Defined by the Streaming Classes . . . 105
Aliased Objects . . . 105
Monomorphic Streaming . . . 106
Polymorphic Streaming . . . 106
Release-to-Release Data Compatibility (RRDC) . . . 107
Data Streams . . . 107
Application Data Interfaces . . . 108

Chapter 12. I/O Stream Classes . . . 109
Overview of I/O Streams . . . 109
Input and Output for User-Defined Classes . . . 109
The I/O Stream Class Hierarchy . . . 109
I/O Stream Header Files . . . 110
Stream Buffers . . . 111
What Does a Stream Buffer Do? . . . 111
Why Use a Stream Buffer? . . . 111
Format State Flags . . . 112
Using Manipulators . . . 112
Simple Manipulators and Parameterized Manipulators . . . 112
ios Methods and Manipulators . . . 112
The I/O Stream Classes and stdio.h . . . 113

Chapter 13. Windows, Menus, and Events . . . 115
Windows . . . 115
Frame Windows . . . 115
Menus. . . . 116
Menu Bars and Pull-Down menus . . . 116
Pop-Up Menus . . . 117
Events and Event Handlers . . . 117
Processing Events Using Handlers . . . 118

Chapter 14. User Interface Controls . . . 121
Controls . . . 121
Static Text . . . 121
Entry Fields. . . . 121
Multiline Edit (MLE) Fields . . . 121
Button Types . . . 121
List Controls . . . 123
Combination Box Controls . . . 124
Slider Controls. . . . 124

Spin Buttons . . . 126
Text-Change Events . . . 126
Canvas Controls . . . 127
Split Canvases . . . 127
Set Canvases . . . 128
Multicell Canvas . . . 128
View Ports . . . 129
Drawing Canvas . . . 130
Notebooks . . . 130
Tab Controls . . . 131
Containers . . . 132
Clipboard . . . 132
Toolbars . . . 134

Chapter 15. Application Resources . . . 137
Resources . . . 137
Window Resources . . . 137
Dialog Templates . . . 137
NLSPATH . . . 138
Dynamic Data Exchange . . . 138
Establishing DDE Conversations . . . 139
DDE Transactions . . . 139
Using Applications, Topics, and Items . . . 140
Specifying Data Formats . . . 140
Understanding the DDE Framework Design . . . 140

Chapter 16. User Interface - Direct Manipulation . . . 143
Direct Manipulation . . . 143
Understanding Drag Items . . . 145
Understanding Drag Item Provider . . . 146

Chapter 17. Graphics . . . 147
2D Graphics Overview . . . 147
Geometry . . . 148
Attributes and Bundles . . . 148
Transformations . . . 148
Modeling . . . 149
Rendering . . . 149
Prerequisite Information . . . 149
Useful Information . . . 149
2D Graphics Features . . . 150
Basic Concepts for 2D Graphics . . . 150
2D Coordinate System and Unit Resolution . . . 151
Numeric and Arithmetic Data Types . . . 151
GCoordinate . . . 152
GDegrees . . . 152
GParametric . . . 152
GIntensity . . . 152
Attributes and Bundles Overview . . . 153
Attribute and Bundle Classes . . . 154
IAttributeState . . . 154
IGrafBundle . . . 154
Fill and Frame Attributes . . . 154
Image Attributes . . . 156
Special Purpose Attribute Bundles . . . 156
IFillBundle . . . 156

IFrameBundle . . . 157
IFillandFrameBundle . . . 157
Color and Pattern Attributes . . . 157
IPaint . . . 157
IBaseColor . . . 159
IMaskPattern . . . 161
Frame Styles . . . 161
IPen . . . 162
Pen width . . . 162
Balance . . . 162
Pen Types . . . 163
IJoint . . . 163
ICap . . . 164
Transfer Modes . . . 164
IColorTransferMode . . . 165
IImageTransferMode . . . 165
Image Sampling . . . 166

Chapter 18. 2-D Graphics Geometry . . . 167
2D Graphics Geometry Overview . . . 167
Geometry Classes . . . 167
Graphic Classes . . . 167
Graphics Class Architecture . . . 168
Computational Geometry Classes . . . 168
Points . . . 169
IGPoint2D . . . 169
IGRPoint2D . . . 170
IGRect2D . . . 170
Point Arrays . . . 170
Path and Area Geometry . . . 171
IGline2D . . . 171
IGCurve2D . . . 171
IGLoop2D . . . 172
IGEllipse2D . . . 172
IGPolygon2D . . . 173
Special Geometry . . . 173
IGArea . . . 173
Special Image: IGImage . . . 174

Chapter 19. 2-D Graphics Modeling . . . 175
Modeling Overview . . . 175
Graphics Class Architecture . . . 175
IMGraphic Class Architecture . . . 176
IMGraphic Models . . . 177
IMGraphic Subclasses . . . 177
IMGraphic Attribute Bundles . . . 178
IMGraphic Transformation . . . 179
IMGraphic Bounds . . . 180
IGraphic Hierarchies and Groups . . . 180
IGraphicHierarchy . . . 181
IGraphicGroup . . . 181
IGraphicGroup Relationships . . . 181
IGraphicGroup Attribute and Transformation Hierarchy . . . 182
IGraphic Hierarchy and Group Iteration . . . 183
IGraphicIterator . . . 184
IGraphicGroupIterator . . . 184

Modeling Examples . . . 184
IMGraphic Basic Components . . . 185
IMGraphic and Corresponding Geometries . . . 185
User-defined IMGraphic Subclasses . . . 185
Complex IMGraphic . . . 186

Chapter 20. 2-D Graphics Rendering Pipeline . . . 187
2D Graphics Rendering Pipeline Overview . . . 187
GrafPorts . . . 188
IGrafPort subclasses . . . 188
IBaseRootGrafPort . . . 189
ILinkedGrafPort . . . 189
IStatefulGrafPort (special class) . . . 189
GrafPort Subclass Notes . . . 189
Special GrafPorts in Open Class . . . 190

Chapter 21. 2-D Graphics Transformations . . . 191
2D Graphic Transformations Overview . . . 191
Useful Information . . . 191
The Transformation Matrix . . . 191
Constructors . . . 192
IGrafMatrix Inquiry Functions . . . 192
IGrafMatrix Mathematical Operations . . . 193
Matrix Transformations . . . 193
Identity . . . 193
Translation . . . 194
Rotation . . . 194
Scaling . . . 194
Matrix Concatenation . . . 194
Perspective Map Transformations . . . 195
Matrix Elements . . . 196

Chapter 22. Multimedia Classes . . . 197
Multimedia Devices . . . 197
Multimedia Device Classes . . . 198
Abstract Device Classes . . . 198
Base Functions for Multimedia . . . 199
Audio Devices . . . 199
Waveform Concepts . . . 200

Chapter 23. Application Control . . . 201
Multi-processing . . . 201
Open Class Threading Model . . . 201
GUI and Non-GUI Threads . . . 202
Data Sharing . . . 202
Locking a Resource . . . 203
Event-driven Synchronization . . . 203
Unshared Thread Data . . . 204
Scheduling . . . 204
New or Enhanced Process/Thread Classes . . . 205
Counted Pointers . . . 205
Reference Counted Objects . . . 205

Chapter 24. Notification Classes . . . 207
Event Notification . . . 207
Notification Solutions . . . 208

Notification Classes 208
Notifiers 209
Observers 209
Interests 209
Notification Event Types 210
The Notification Family of Classes 210
Implement a Notification 210
Asynchronous or Synchronous Notification 211
Multi-threaded Notification 211

Chapter 25. Error Handling and Testing - Exceptions 213
The Exceptions Mechanism 213
Using Exceptions. 213

Chapter 26. Application Testing. 215
Overview of Application Testing 215
Test Framework Components 215
  Test Classes 215
  Runtest macros 215
Test Framework Architecture 216
The ITest Base Class 216
The Decision Function. 217
  The Decision Function Overview 217
  The ITestCollection Class. 217
  The ITestMultiplexer Class 217
Timing Tests 218
  ITimingTest 218
  IStartStopTimingTest 218
Structure of Loops 218
Accuracy. 219
Tolerance 219
Success and Failure 220
Precision. 220
Protocol Tests 221
Auxiliary Classes. 222
  ITieredTextBuffer 222
  IArgumentDictionary 222
Test Macros 223
  runTestImplementationMacro 224
  beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro,
    runTestResultMacro 224

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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

IBM may change this publication, the product described herein, or both.

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 IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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.

Programming Interface Information

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

General-use programming interfaces 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.

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
DATABASE 2
DB2

© Copyright IBM Corp. 1998

IBM
Object Connection
Operating System/2
OS/2
OS/400
Presentation Manager
SAA
Systems Application Architecture
TeamConnection
VisualAge
Workplace Shell

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

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

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

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

PC Direct is a registered trademark of Ziff Communications Company and is used by IBM Corporation under license.

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

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

Industry Standards

VisualAge C++ 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 1003.1-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About this Book

This Portable Document Format (PDF) file is a collection of some of the concepts and tasks selected from the online information for this product. PDF files give you an easier way to review or print a group of related topics.

References to VisualAge C++ in this book should be interpreted as VisualAge C++, Version 4.0.

© Copyright IBM Corp. 1998

Chapter 1. Introduction

Introduction to the Open Class Library

This introduction to the Open Class Library lists the frameworks and libraries included in this release; it specifies the navigation route to the Open Class Library documentation and describes graphics conventions. This documentation assumes that you are proficient with both the C++ programming language, and with object-oriented programming concepts and techniques.

Frameworks and Classes

To assist you in learning about the classes and to guide you as you start developing portable applications, we have organized the classes and frameworks into the following basic categories:

- Application Control
- Collections
- Error Handling and Testing
- Graphics
- Math
- Multimedia
- Streams and Storage
- Text and Internationalization
- User Interface

In addition, we are providing for the first time, the ANSI C++ Standard library. The Open Class library is not yet using the Standard library internally, and similar function libraries are not compatible (such as the Iostreams classes and the ANSI C++ I/O library).

Here is a brief summary of the contents of each of these categories:

- The Application Control classes provide support for multi-threaded execution environments.
  - Processes classes support the creation and manipulation of processes.
  - Threads classes support the creation and manipulation of threads.
  - Reference Counting classes manage thread-safe access to multiply referenced objects.
  - Notification classes support notifying interested objects of changes in other objects.
- The Collection Classes provide a set of commonly used abstract data types used to build collections. Collections can have properties such as sorted or unsorted; ordered or unordered; unique-element or multiple-element.
- The Error Handling and Testing classes provide support for building robust and well-behaved applications.
  - Test Framework creates and runs unit tests in the Open Class environment.
  - Exceptions classes provide support for detecting and conveying information about unusual circumstances in applications.

© Copyright IBM Corp. 1998

- The 2D Graphics Framework supports the platform-independent creation, manipulation, and rendering of 2D graphic objects based on patented mathematical algorithms. The framework includes these class groups:
  - Geometry classes define forms for path, area, and image geometries.
  - Attributes describe graphic characteristics such as paint, pen and joints, that let you create groups of attributes to apply to graphic objects when rendering them.
  - Transformations let you apply mathematical transformations such as scaling, rotating, and translating to 2D graphic objects.
  - Modeling classes combine geometry objects with attributes and transforms for rendering and storing graphics and creating hierarchies of graphics.
  - Rendering pipeline classes implement the rendering of 2D graphic objects.
- The Math classes define objects, operators and syntax which create, manipulate, and translate values set into short, int, long, float, double, complex, and scientific formats and provide support for binary coded decimal format. These classes will be deprecated in favor of the new STL Class Numerics library in a future release.
- OS/2 WIN The Multimedia classes support the creation of applications that integrate text and graphics with a combination of audio, motion video, images, and animation. Using the multimedia classes and the window control classes, you can implement an interface for your application that looks like the controls of common electronic devices, such as stereo components and video cassette recorders (VCRs). Your application can use these controls as interfaces to control audio and video media that is presented to the user. Support for media devices is abstracted into classes that contain the data and functions essential for the operation of the real-world devices that they model. This framework includes these device classes groups from which you can directly instantiate device objects:
  - An audio amplifier-mixer class
  - A CD audio player class
  - A CD Extended-Architecture player class
  - A Digital video player class
  - A Master audio class
  - a MIDI sequencer class
  - A Waveform audio player class
- The Streams and Storage classes implement persistent storage mechanisms for Open Class components, and includes:
  - Streaming classes that support streaming data in C++ objects in and out of persistent storage.
  - The IOStreams Classes create and manage predefined and user-defined streams objects, which transfer data among files, buffers and peripherals. These classes will be deprecated in favor of the new STL Class library in a future release
  - File System classes, which provide portable abstractions that allow you to manipulate physical file system entities such as volumes, directories, and files.
- The Text and International Frameworks provide support for Unicode text strings and for building easily localizable components. These frameworks include the following components:
  - The Unicode support classes, which support inquiries about the stylistic and symantic properties of characters, character sets, and scripts (writing systems).
  - The Text display classes support user editing, and display and storage of text.

  - The International frameworks support the creation of international applications and manipulation of international text, including language-sensitive comparison of text strings, conversion between character sets, and a locale mechanism for access to portable and host-specific resources.
- The User Interface classes provide support for building the graphical user interface of an application.
  - Base Window, Menu, Handler, and Event classes provide support for the basic windows, handlers, events, and menus used by the applications you develop. These classes encapsulate the basic graphical building blocks that are used to construct application windows. This encapsulation allows window position and appearance (windows and menus) to be separated from event-handling (handlers and events).
  - Standard Control Classes provide support for the standard controls such as entry field, static text, and buttons used by the applications you develop.
  - Advanced Control, Dialog, and Handlers Classes provide support for the advanced controls such as container, notebook, tool bar, and the font and file dialogs used by the applications you develop.
  - Direct Manipulation Classes provide support for the direct manipulation used by the applications you develop.

Changes in this Release of Open Class

The sections that follow describe the major changes in this release and identify deprecated classes and functions.

AIX Changes

These features are now implemented on AIX:

- Accelerators
  Support for accelerators existed in the previous AIX release, although in a more limited form.
- Clipboard
- Direct Manipulation
- Fly-Over Help
- Resources
- Toolbar
- Handling of Mouse Movement Events
- Portable Font Dialog
- 2D Graphics

Toolbar and Related Changes

New Styles for Direct Manipulation Support

For performance reasons, dragdrop support is no longer enabled by default. This change necessitated the addition of a new style and deprecation of an existing style. The style IToolBar::dragdrop enables dragging and dropping of the tool bar items. This style is not part of classDefaultStyle. It replaces the style IToolbarButton::noDragDrop, which has been deprecated.

The style IToolBarButton::dragDelete enables the dragging and dropping of toolbar buttons to the shredder; the drag handler queries this style. This style is part of classDefaultStyle. It replaces the style IToolbarButton::noDragDelete, which has been marked as deprecated. Specifying this deprecated style has no effect. Although buttons could originally be dropped on the shredder, to get this style, you must now use the classDefaultStyle or specify the style explicitly.

There is a new override of IToolBar::layout.

There is a new performance setting, IPerformanceSettings::dynamicToolBarButtons. The setting effects the behavior of the following toolbar button functions:

- IToolBarButton::setStandardBitmapSize()
- IToolBarButton::setStandardTextLines() and
- IToolBarButton::setStandardTextWidth()

With the setting enabled, these functions will dynamically change the settings of all IToolBarButtons created with the IToolBarButton::standardFormat style. Calling these functions causes existing tool bar buttons to adopt the new settings immediately. Each button will resize and redraw with the new settings.

Without this setting enabled, calling these functions causes default settings to be used during the creation of any toolbar buttons that are subsequently created with the IToolBarButton::standardFormat style. This does not affect existing toolbar buttons. By default, this setting is not enabled, which provides for better tool bar performance.

New Default Tool Bar Buttons

The new Lotus toolbar buttons with Open Class Library is included with this product.

IToolBar::Style and IToolBarContainer::Style changes

IToolBar::Style and IToolBarContainer::Style has been changed so that you can no longer combine style objects of these types with ISetCanvas::Style objects. Today, if you do specify an orientation or alignment via ISetCanvas::Style objects when constructing an IToolBar or IToolBarContainer, these values are essentially ignored because both IToolBar and IToolBarContainer explicitly set the orientation and alignment themselves. The only ISetCanvas styles that you can effectively set are the packing styles. The packing styles can also be set by calling ISetCanvas::setPackType, and ISetCanvas::decksByGroup. which IToolBarContainer gives you by default.

The ability to use IWindow, ICanvas, and ISetCanvas styles on the IToolBar constructor has been deprecated. IToolBar currently does not pass IWindow styles to the public ISetCanvas constructor it calls, and it essentially ignores extended styles. This is because it explicitly configures the ISetCanvas using function calls. Using application-specified IWindow, ICanvas, and ISetCanvas styles could break existing code. This behavior also applies to IToolBarContainer::Style.

Transparency in Tool Bars

To optimize performance, this product no longer supports transparency by default. The product added the IToolBarButton::transparentBitmap to provide trasparency support. Here are the two places in which transparent bitmaps are used within toolbar buttons:

- One use is to display the button in an up state. The transparent area of the bitmap shows through to the background color, which can be changed by the user.
- The other use is to display a button in latched state. In latched state, a button is drawn in the down state, with the transparent area of the bitmap showing through to a half-toned background look.

To avoid transparency overhead, this product introduces a new style for IToolbarButtons buttons called IToolBarButton::transparentButton. This style is not set on as the default.

For the IOC supplied bitmaps (IC_ID_OPEN, etc), this product supplies non-transparent bitmaps, and skips any transparency related code. If you want to use the existing transparency function, this product ships the existing IOC bitmaps under new IDs, such as IC_ID_OPEN_WITH_TRANSPARENCY, and requires you to use the new bitmap IDs and set the IToolBarButton::transparentButton style for the button. For user supplied bitmaps, the default is that transparency is turned off. If you want transparency and you supply a transparent bitmap, you must turn the style on for those buttons.

The transparentButton style affects both buttons drawn in an up state and those drawn in a latched state. Without the style set for a button, if the button is drawn in the latched state when the button view is bitmap only, you see the non-transparent latched look. The button is painted as if in the down state, with a slight half-toned look just inside the border and around the bitmap.

When the button view is bitmap and text, the button background color around the text area, which you can configure, might not match the non-transparent gray of the bitmap. In this case, you have the option to use a transparent bitmap for up state and latched buttons. You must make this choice at button creation, or use the function, IToolBarButton::enableTransparency(bool enable = true).

IWindow Changes

Reparenting of windows is not supported on Motif. This is a Motif-only restriction requiring that any window added to a toolbar must be a child of that toolbar.

OS/2 WIN You can still add previously created windows to a toolbar on OS/2 and Windows since IOC will reparent the window automatically.

Canvas Classes

This product added the following function to ICanvas to determine if windows have been added to the canvas.You may need to override this function in your own ICanvas derived classes.

```
virtual bool
  hasChildrenToLayout ( ) const;
```

New Protected Constructor for ISetCanvas

This product added the following protected functions to ISetCanvas to simplify the construction of derived classes and support delayed window creation:

```
ISetCanvas ( );
ISetCanvas
&initialize ( unsigned long windowId,
     IWindow* parent,
     IWindow* owner,
             const IRectangle& initialRect,
     unsigned long style,
             unsigned long extendedStyle );
```

Calling the protected constructor does not create a presentation system window. However, you can create a canvas presentation system window by calling initialize. The style arguments are of type unsigned long so that styles defined by a class derived from ISetCanvas can also be used.

IToolBar and IToolBarContainer call this protected constructor, which gives a set canvas with a default configuration but no underlying presentation system window. IToolBar and IToolBarContainer later call ICanvas::initialize to create the presentation system window.

Border Support

ICanvas now has the capability to draw a border with text. Previously, ISetCanvas was the only canvas class with this support. ICanvas provides the following public members to provide this support:

```
static const Style
 ICanvas::border;
static INotificationId const
 ICanvas::borderTextId;

virtual ICanvas&addBorder ( ),
 &removeBorder ( ),
 &setBorderText ( const IText& borderText, bool showBorder=true),
 &setBorderText ( const IResourceId& borderText, bool showBorder=true),
 &setBorderColor ( const IColor& borderColor),
 &resetBorderColor ( );

virtual bool
 hasBorder ( ) const;
IText
 borderText ( ) const;
 virtual IColor
borderColor ( ) const;
```

Notes:

- Constructing a Canvas with the style ICanvas::border creates a canvas with a border but no text.
- Calling ICanvas::setBorderText(text) adds and displays a border (and the corresponding border style) with text
- Calling a canvas containing border text with ICanvas::setBorderText(0) causes a canvas to remove its border text, but leaves the border.
- Calling removeBorder removes the border and any border text and repositions its children to use the additional space (the canvas does not draw its border text).

You can query and set the color or font of the border text using the IText class.

The following protected members were added to ICanvas for derived classes to implement layout within a border:

```
IPoint
 topLeftLayoutOffset ( ) const,
 bottomRightLayoutOffset ( ) const;
IRectangle
 rectangleInsideBorder ( const ISize& sizeWithBorder ) const;
ISize
 sizeWithBorder ( const ISize& sizeWithoutBorder ) const;
```

The following member from ISetCanvas was removed:

```
static const Style
ISetCanvas::border;
```

The following member of ISetCanvas has changed its behavior::

```
ISetCanvas::setText(); // if specifying text, change to call setBorderText(text, true), else
       // change to call removeBorder.
ISetCanvas::text()    // change to call borderText
```

The following member of ISetCanvas was deprecated:

```
static INotificationId const
ISetCanvas::textId; // Replaced by ICanvas::borderTextId.
```

Explicitly Adding and Removing Child Windows from ISetCanvas

An interface that allows you to add and remove child windows from the layout of a set canvas was added. This feature allows:

- A tool bar (which is derived from ISetCanvas) in Motif to filter misfit controls and remove child controls which it cannot do currently given the lack of IWindow::setParent.
- An application a way to temporarily remove a control from the layout of a set canvas in Motif (an application usually does this today in OS/2 by reparenting the control to the object window).
- IToolBar to behave predictably for the case of a control that has not explicitly been added to the tool bar

To simplify layout processing in ISetCanvas, tthe application should specify at construction time whether it is using the add/remove interface. Use the following style to do this:

```
 ISetCanvas::explicitAddsNeeded
```

The following interface for the application to add/remove child windows to the layout of the canvas was added:

```
virtual ISetCanvas
 &add ( IWindow* child ),
 &remove ( IWindowl* child),
 &replaceChildWith ( IWindow* existingChild IWindow* newChild );
bool
 areExplicitAddsNeeded() const
```

Notes:

- For a set canvas to manage the size and position of a child window, that child window must be considered to have been added to the canvas. Not using the style causes all child windows initially to be considered added. Calling the add function to add an already added child window produces no result. Specifying the style causes all child windows initially to be considered not added; in this case you must call add to add a child window to the layout of the canvas. Whether you use the style or not, you can still remove/replace child windows; you can add

back removed/replaced child windows. See the isInLayout function for more details on when a child window is considered added or not added.

- When adding a child window, ISetCanvas also shows and enables the child. However, ISetCanvas will not change the z-order of the child window. The z-order will still determine where the child window appears in the layout.
- When removing or replacing a child window, the ISetCanvas will hide the child window and size it to (0,0). Not hiding the child window causes the child window to remain visible to the user.
- When replacing a child window, ISetCanvas moves the new child to the existing child's position in the z-order. It copies the tab stop and group styles of the existing child to the new child. It also shows the new child. All of this transpires before the old child is removed.

To query whether a set canvas is managing the size and position of a child window, the following was added:

```
virtual bool
 isInLayout ( IWindow* child ) const;
```

This function returns true for a child window if you

- Added the child window to the canvas via a call to ISetCanvas::add or replaceChildWith, and did not remove or replace it afterwards by calling remove or replaceChildWith.
- Did not create the canvas with the style ISetCanvas::explicitAddsNeeded, and never specified the child window on a call to ISetCanvas::add, remove, or replaceChildWith.

This function returns false for a child window if you

- Removed or replaced the child window by calling remove or replaceChildWith without later adding it back to the canvas.
- Created the canvas with the style ISetCanvas::explicitAddsNeeded, and never specified the child window on a call to ISetCanvas::add and never added it to the canvas by calling replaceChildWith.

IViewPortPerformance

The following added interface improves IViewPort performance by allowing the viewport child (view window) to be set:

```
virtual IViewPort
 &setViewWindow ( const IWindowHandle& viewWindow );
```

This portable interface boosts IViewPort performance on both OS/2 and Windows by avoiding the child window lookup currently required to find the view window to scroll on each scroll message. This interface also allows you to easily change the window that a view port scrolls.

Note: The restriction of allowing only a single view window per viewport is removed and an exception is no longer thrown when mutiple view windows exist. If setViewWindow is not called, the first view window found is used as the viewports' view window. In order to replace a view window with a different view window, you must issue setViewWindow. The previous view window is hidden.

IWINDOW Changes

Window Owner versus Parent

The usage of an owner different from the parent is discouraged in child windows. In top level windows (IFrameWindows, typically), owner and parent remain distinct and are supported portably as such. A top level window owned by another top level window remains above its owner in the Z order and will be closed when its owner is closed. Modality is also with respect to the owner. In IFrameWindow, the parent window is the one that is less portable. An IFrameWindow usually has a parent of 0 or IWindow::desktopWindow, which is the portable variant. Setting the parent of an IFrameWindow to another window can be useful in some scenarios but is less portable. In particular, an IFrameWindow child of another IFrameWindow will behave differently on the three platforms.

From an API perspective, there is no change - constructors which took both a parent and an owner continue to do so, even if the owner is ignored. This product does not provide parent-only-required constructors. In essence, the change here is one of documentation. The additional parameters are not needed and will be ignored but behavioural changes may occur on OS/2 if specific behaviour was expected when the parent was different from the owner.

Changes toIObjectWindowand IWindow for Support of 'bound' Windows

In previous releases, the notion of a primary window as defined by the operating system, and the concept of a window requiring that the message queue persist for the lifetime of that window, were treated as a single concept. Both concepts are useful, and they are now treated as distinct in the interface, as follows:

- The behavior of IWindow::setParent and setOwner has changed. IWindow::setParent and setOwner has changed so their checks for setting the primary window flag matches IWindow::startHandlingEventsFor. This breaks any code that relies on using IWindow::setParent to remove the primary window flag. A specific interface for being able to control the primary windowflag has also been added.
- The behavior of IWindow::isPrimary has been changed so that it only returns true if the window is truly a primary window.
- A new style has been be added to IObjectWindow, boundToMessageQueue. The default style is boundToMessageQueue.
- A new protected function has been added to IWindow to bind the messagequeue. The IObjectWindow constructor must call this function if the style is present.
- The 3 places that the primary flag can be set (setParent, setOwner, andstartHandlingEventsFor) should also set the 'bound' flag as appropriate. All primary windows are also 'bound' windows, but all 'bound' windows are not primary.
- 'Bound' windows are now checked instead of primary windows when deciding whether to shut down the message queue.

/* New interface in IObjectWindow */

```
public:
class Style;

IObjectWindow ( const IObjectWindow::Style& style = defaultStyle() );
INESTEDBITFLAGCLASSDEF0( Style, IObjectWindow );

static const Style
 IC_IMPORTU classDefaultStyle,
 IC_IMPORTU boundToMessageQueue,
 IC_IMPORTU noStyle;

static Style
 defaultStyle ( );
```

```
static void
 setDefaultStyle ( const Style& style );
private:
INESTEDBITFLAGCLASSFUNCS(Style, IFrameWindow);
static Style
 fCurrentDefaultStyle;
     /* New interface in IWindow */
     protected:
     IWindow
      &bindMessageQueue ( );
     bool
      isBoundToMessageQueue ( ) const;
```

In several classes (the canvases), collections of objects are created that are nothing more than a pointer to an IWindow and a bit of extra data needed by the class that is specific to the IWindow. These collections are needed because there is no way to extend the data of an IWindow from outside the object. Keeping these collections of objects have their own problems because they must be kept up to date so that they accurately represent not only the child windows of a parent object but they typically need to do so in sibling order. This requirement is difficult enough that ISetCanvas rebuilds the collection each time it does layout so that it is accurate.

Extensions to the data of an IWindow are now allowed. IWindow is updated to store an array of pointers to objects. Any class that needs to store data in an IWindow object first queries to receive a magic token which it uses to store and retrieve data from an IWindow object (the magic token is implemented as an index into the array of pointers). Instead of creating collections of objects to represent additional data, classes like the canvases define and create an object that they store as additional IWindow data. They then use IWindow::ChildCursor to walk through the IWindow children and retrieve their extended data. Support has been added to IWindow::ChildCursor to allow it to visit only IWindow children.

The following classes and members have been implemented:

- IWindow::DataHandle
  This is the magic token.
- IWindowData
  Abstract base class of data added to an IWindow. It exists to provide a virtual destructor so that IWindow can manage the destruction of the object. The idea is that IWindow adopts the object and deletes it in the IWindow destructor.
- static DataHandle IWindow::dataHandleWithKey(const char* dataKeyName);
  This thread-safe function increments a counter so that each new dataTypeName gets a different offset. It also keeps a collection of registered names so that duplicate requests return the same DataHandle.
- inline IWindowData* IWindow::windowData (const DataHandle& typeToken) const;
  This inline function indexes the array (stored directly in IWindow) and returns the pointer at the offset of the passed DataHandle.
- IWindow& IWindow::adoptWindowData (const DataHandle& typeToken, IWindowData* windowData);
  This member checks to see if an array of IWindowData exists for the window and allocates it if it doesn't. It also checks to see if the array is large enough to contain the index and reallocates it if necessary. It then stores the pointer in the array. If an entry already exists at the handle location, it is deleted before the new window data is stored.

- IWindow::ChildCursor::ChildCursor(IWindow& parent, bool onlyIWindowChildren = false);
  This function is updated to allow cursoring only IWindow children. When cursorIWindowChildren is true, the cursor movement functions call windowWithHandle to query and store the IWindow* at the cursor location. The cursor movement function continues to enumerate and call windowWithHandle until all children have been visited or windowWithHandle returns a valid IWindow pointer.
- IWindow* IWindow::childWindowAt (const ChildCursor&) const;
  This function returns the IWindow* stored in the child cursor. The function first checks the validity of the cursor (IASSERTSTATE(fWindowHandle != 0)). If the child cursor's IWindow* value is non-zero, it returns that value. If the cursor's IWindow* value is zero it returns the result of windowWithHandle (which may be zero, no exception is thrown).

A new version of IWindow::create that takes an IWindow* for parent and owner instead of IWindowHandles has been added. Use this new create when you subclass an IWindow object. The library then ensures that the correct handle for the creation (parent->handleForChildCreation) of the control is used.

The use of handle on Motif was different from its use on Windows and OS/2. If an aggregate could have children, such as IFrameWindow and IViewPort, handle was used as the parent of the child. Windows and OS/2 get around this problem by using a setParent to change the parent of the child to the correct control. Motif can not use this technique because reparenting is not available. Motif had a topHandle and a handle so that children of a control can have a different parent handle than the handle used for geometry management. In Motif, the topHandle was the handle of the control (widget) that encompassed all of the widgets of the aggregate. The topHandle was used for geometry management of the control such as moving and sizing the control. The problem with topHandle is that it was Motif only and had the same meaning as handle does on OS/2 and Windows.

A new member function returns the handle that a child of a control should use as their parent. When a control is created and an IWindow* is specified as the parent, handleForChildsParent is used instead of handle. The default implementation of handleForChildsParent just returns handle. Any subclass that needs a different implementation, like IViewPort, overrides this member function and returns the appropriate handle.

Here are the changes to IWindow:

```
#ifdef IC_PMWIN
virtual IWindowHandle
 create ( unsigned long id,
  const char* text,
          unsigned long style,
          const char* windowClass,
          const IWindowHandle& parent,
  const IWindowHandle& owner,
          const IRectangle& initRect,
          const void* ctlData,
          const void* presParams,
          IWindow::SiblingOrder ordering = defaultOrdering(),
          unsigned long extendedStyle = 0 );
virtual IWindowHandle
  create ( unsigned long id,
           const char* text,
           unsigned long style,
           const char* windowClass,
           const IWindow* parent,
```

```
            const IWindow* owner,
            const IRectangle& initRect,
            const void* ctlData,
            const void* presParams,
            IWindow::SiblingOrder ordering = defaultOrdering(),
            unsigned long extendedStyle = 0 );
#endif

#ifdef IC_MOTIF
virtual IWindowHandle
create ( unsigned long id,
 const char* text,
 unsigned long style,
 IXmCreateFunction createFunction,
 const IWindowHandle& parent,
 const IWindowHandle& owner,
         const IRectangle& initRect,
         const void* callerArgList,
         const unsigned int callerNumberArguments,
         IWindow::SiblingOrder ordering = defaultOrdering(),
         unsigned long extendedStyle = 0 );

virtual IWindowHandle
create ( unsigned long id,
 const char* text,
         unsigned long style,
         IXmCreateFunction createFunction,
         const IWindow* parent,
         const IWindow* owner,
         const IRectangle& initRect,
         const void* callerArgList,
         const unsigned int callerNumberArguments,
         IWindow::SiblingOrder ordering = defaultOrdering(),
         unsigned long extendedStyle = 0 );
#endif //IC_MOTIF
```

The implementation of the new create just uses the old create passing through the correct handles.

For example:

```
| IWindow::create |
| This create will take care of the using correct parent's handle for the creation of children.

    ----------------------------------------*/

IWindowHandle IWindow::create( unsigned long id,
const char* text,
unsigned long style,
IXmCreateFunction createFunction,
const IWindow* parent,
const IWindow* owner,
const IRectangle& initRect,
const void* callerArgList,
const unsigned int callerNumberArguments,
IWindow::SiblingOrder ordering,
unsigned long extendedStyle )

    {
    return this->create( id,
    text,
    style
    createFunction,
    parent->handleForChildsParent(),
```

```
owner->handle(),
initRect,
callerArgList,
callerNumberArguments,
ordering,
extendedStyle  );
}
```

Summary

Motif

topHandle is replaced with handle ; topHandle is deprecated . Do not call addRelated or rely on windowWithHandle working on anything but the registered handle.

All Platforms
Override handleForChildCreation if you are an aggregate that can have children. Call IWindow::create with your parent's handleForChildCreation (instead of handle) or use the new IWindow::create override.

Keyboard

The following data types are deprecated as they provide little utility and are nonportable:

```
public:
#ifdef IC_MOTIF
static unsigned long
IC_IMPORTU ulShiftMask,
IC_IMPORTU ulAltMask,
IC_IMPORTU ulCtrlMask;
#endif
protected:
#ifndef IC_WIN_FLAGNOP
static const unsigned long
IC_IMPORTU ulCharacterFlag,
IC_IMPORTU ulScanCodeFlag,
IC_IMPORTU ulVirtualFlag,
IC_IMPORTU ulRepeatFlag,
IC_IMPORTU ulUncombinedFlag,
IC_IMPORTU ulShiftFlag,
IC_IMPORTU ulCtrlFlag,
IC_IMPORTU ulForCompositeFlag,
IC_IMPORTU ulCompositeFlag,
IC_IMPORTU ulInvalidCompositeFlag;
#endif
static const unsigned long
 IC_IMPORTU ulUpTransitionFlag,
 IC_IMPORTU ulAltFlag;
```

Frame Window

Users who wish only to set the title text on a frame window now have a simple, straightforward, portable way to do this. You no longer have to create an ITitle object to have title text on a frame window.

The following new functions have been added to IFrameWindow:

```
virtual IFrameWindow&
 setTitleText ( const IString& titleText ),
 setTitleText ( const IResourceId& titleResId );
virtual IString
 titleText ( ) const;
```

A notification to IFrameWindow for text changes has also been added:

```
static INotificationId const
IC_IMPORTU titleTextId;
```

Inheritance

Classes that inherit from either IBase or IVBase no longer exist.

These new classes and functions have been added::

- Provide a way to ask for Helvetica 30, such that the font looks the same as in other applications.
  The font will also scale (on the screen) the same way fonts scale in other applications as device resolution is changed.
- Provide a way to program in pixels, if you want to.
- Provide a way to achieve WYSIWYG (metric fidelity), if wanted.
- Provide a way to create objects so they can be freely streamed between machines and address spaces, and they mean something useful on the other side of the stream.

There is a new member function on a GrafPort, that allows you to get the transformation matrix that converts from GCoordinates to pixels, for example:

```
aPort.GetDeviceTransform(IGrafMatrix &aGrafMatrix);
```

However, it is not necessary to know this in order to convert back and forth between ideal coordinates and device coordinates (pixels).

Summary of new function:

- Use IMetricGrafPort, if you want to program in GCoordinates, and you don’t want to deal with conversions between pixels and GCoordinates yourself.
- In both cases, if you want Helvetica 30 point text, use IPointSizeStyle(30.0).

Color

New color support has merged with the existing IColor class. The following changes have occurred:

- The enums Color and SystemColor have been deprecated in favor of new enums called called EPredefinedColor and ESystemColor. The enum values have been changed from 'white' to 'kWhite', and so on.
- IColor still contains an optional system color enum. This minimizes breakage for your existing applications.
- When the ESystemColor enum is used, it means always ask the OS what color this is, before returning red, green, or blue values from getters.
- This enum is streamed. Use of the enum is not portable across systems or display devices.
  The platform will make a best effort to interpret the value appropriately.
- This product has replaced void setRed(CharIntensity) by the upwardly compatible function
  IColor& setRead(CharIntensity). This permits chaining.

- The new IColor contains all the member functions of the old IColor and the old IBasicColor. There are no conflicts, except for those mentioned above.

The IBaseColor class had replaced IBasicColor and IColor became a subclass of IBaseColor. IBasicColor still exists for backward compatibility.

Threads

A GUI thread class and a non-GUI thread class are now available. The new classes are:

- ICurrentNonGUIThread
- ICurrentThread

Menus

In the past, you could call any IWindow function on an IMenu even though the function did not do anything under Windows. For this release, you can only call IWindow methods by first calling IMenu::window() to get the underlying window; this method is only supported under OS2 and Motif. Selected IWindow methods have been added to IMenu or one of its derived classes and implemented there in a portable way. The menu hierarchy has changed. IMenu no longer derives from IWindow.

foregroundColor and backgroundColor have been added to the new menu headers. Since these two colors on Win32 actually apply to all menus in the system, this product does not support setXXXColor.

Version Number

IC_MAJOR_VERSION is set to 360 and IC_OBSOLETE is set to 400 for IC_PM, IC_WIN, and IC_AIX. IC_MINOR_VERSION is set to 0.

ICONTROL

The group and tabStop styles and their associated functions have moved from IControl to IWindow.

Container

New interface forIContainerObject

The following functions set a special in-use icon to be used for this particular object in all containers it exists in:

- virtual IContainerObject& IContainerObject::setInUseIcon(const IPointerHandle& inUseIcon);
- virtual IContainerObject& IContainerObject::setInUseIcon(const IResourceId& inUseIconId);
- virtual IContainerObject& IContainerObject::setInUseIcon(unsigned long inUseIconId);
- virtual IPointerHandle IContainerObject::inUseIcon( ) const;

The behavior is as follows:

- On OS/2 and Windows PM compatible:
  When the client calls setInUse() the code checks to see if a special in-use icon

has been set for the object. If so, this icon will be displayed, the original icon cached, and the normal processing will continue. This includes sending the container a CM_SETRECORDEMPHASIS message to draw the hatched background behind the icon. When removeInUse() is called, the previous icon is cached will be replaced and the normal processing will continue. This includes removing the hatched background. If a special in-use icon is not set for the object, the normal processing will take place, namely drawing of the hatched background.

- On native Windows and Motif:
  When the client calls setInUse() the code checks to see if a special in-use icon has been set for the object. If so, this icon will be displayed and the original icon cached. If there is not a special in-use icon set, this call will do nothing. When the client calls removeInUse() the previous icon is cached, if any, will be replaced. This product should not provide the hatched background because it is an OS/2(Windows PM Compatible) specific behavior.

Graphics

The 2D graphics provided in VisualAge for C++ Version 3.0 and 3.5 has been deprecated.

ICoordinateSystemand Application Origin:

To provide the capability to write portable applications, this product does the following:

1. If an application does not call ICoordinateSystem::setApplicationOrientation(), then the orientation defaults to upper-left. This behavior differs from previous releases, where the application
   orientation was upper-left for Windows and lower-left for OS/2.
2. For the AIX release:
   - If an application does call setApplicationOrientation(), then the default constructor of IExtendedRootGrafPort (the one that doesn't specify an orientation) will use that orientation.
   - Calling ICoordinateSystem::setApplicationOrientation() in the middle of your application, after IExtendedRootGrafPort's have been created results in undefined behavior. IExtendedRootGrafPort just looks at the ApplicationOrientation once.
   - The enum ICoordinateSystem::Orientation is deprecated.
   - A new enum, ICoordinateSystem::EOrientation() has been created. It has three values: kOriginUpperLeft, and kOriginLowerLeft, and kDefaultAppOrientation for default value purposes.

Examples:

```
function() {
ICoordinateSystem::setApplicationOrientation(ICoordinateSystem::kOriginLowerLeft);
IExtendedRootGrafPort aPort(hps); // origin defaults to appOrientation, lower left
}

function() {
ICoordinateSystem::setApplicationOrientation(ICoordinateSystem::kOriginUpperLeft);
IExtendedRootGrafPort aPort(hps); // origin defaults to appOrientation, upper left
}
function() {
IExtendedRootGrafPort aPort(hps); // origin defaults to upper left
}
function() {
IExtendedRootGrafPort aPort(hps, ICoordinateSystem::kOriginUpperLeft); // origin always upper left
       // regardless of SetApplicationOrientation()
```

```
}
function() {
IExtendedRootGrafPort aPort(hps, ICoordinateSystem::kOriginLowerLeft); // origin always lower left
       // regardless of SetApplicationOrientation()
}
```

Combobox

The current combo box can not drop down past its parent's area. This limitation has been removed by parenting the listbox portion of a drop down combo box to a shell. This enhancement has been implemented to support toolbars. This is an AIX-specific enhancement (it already behaves this way on OS/2 and Windows).

Boolean Type Definition

Boolean has been changed to bool to meet ANSI standards.

Scroll Bar

The IScrollBar class currently contains the autoSize style. This style is supported on PM and Motif, but is not supported in Windows. In PM, specification of the autoSize style results in a vertical scrollbar that will be automatically sized to the system scrollbar width if a width of 0 is specified, or a horizontal scrollbar that will be automatically sized to the system scrollbar height if a height of 0 is specified. In Motif, this style was implemented in the class library to always force the scrollbar width (height) to the (hardcoded) system default dimension.

The IScrollBar::autoSize style in PM has been deprecated and removed from Motif for the following reasons, which assume vertical scrollbar:

- In order to use it in PM, a width of 0 must be specified when the scrollbar is sized. This style, therefore, has no value for a scrollbar in a multicell canvas since the canvas will use the minimum size to determine the optimal width and will size the scrollbar accordingly. If the scrollbar is not constructed as a child of a multicell canvas, then the system width needs to be used in order to properly position it and other child windows. Whatever width is assumed in order to acquire the preferred layout should actually be used to size the scrollbar.
- The Motif implementation was not equivalent to the functionality provided in PM. In order to provide the similar functionality, this product now needs to look for ConfigureRequest events when the autoSize style is present. If the requested width is 0, we will need to change to the system scrollbar width.
- A scrollbar can be constructed using the system scrollbar width as returned by the static IScrollBar::systemScrollBarWidth function.

ICUSTOMBUTTON

IToolBarButton inherits from ICustomButton. The default protected ICustomButton constructor attaches a default ICustomButtonDrawHandler handler. The current implementation of IToolbarButton makes the ICustomButton default handler unnecessary. IAnimatedButton also uses this ICustomButton constructor and also does not need the default ICustomButtonDrawHandler. Each tool bar button incurred the overhead of the additional handler attached to it. For performance reasons, this handler has now been removed and the following changes made. This product has

- Added an ICustomButton protected function called removeDefaultHandler and a corresponding addDefaultHandler. This is consistent with IFrameWindow, which includes these functions.
- Changed the behavior of the ICustomButton protected constructor so it does not add the ICustomButtonDrawHandler.
- Added two new protected functions: addDefaultDrawHandler(); and removeDefaultDrawHandler();

You cannot override the default paint handler for custom buttons. Use the two new functions or if the class derived from ICustomButton wants to use the default paint handler, call the addDefaultDrawHandler function to add the handler. This is a change in behavior because such a class used to get the handler added automatically. However, a derived class that has its own paint handler derived from the ICustomButtonDrawHandler will not see any behavior changes and might see some performance improvements.

IMOUSEPOINTERHANDLER

This class is ignored on Motif (it will override no event-handling callback functions). IMousePointerHandler still overrides mousePointerChange on Windows and OS/2.

New Classes

IManagedPresSpaceHandle

Due to the circular dependencies between Open Class and the new graphics classes, this product has removed IWindowHandle from the interface of graphics and replaced it with a new class, IManagedPresSpaceHandle.

IManagedPresSpaceHandle puts the responsibility for managing the lifetime of the handle on the client. The class contains constructors, from both an IWindow* and const IWindowHandle&, that acquire the handle. It also contains a destructor to free the handle and an operator IPresSpaceHandle so an object of the class can be passed to any function that takes an IPresSpaceHandle.

This new class is implemented in imngpres.* and the new interface is as follows:

```
class IManagedPresSpaceHandle
{
  public:
  IManagedPresSpaceHandle ( IWindow* window);
  IManagedPresSpaceHandle ( const IWIndowHandle& windowHandle);
  ~IManagedPresSpaceHandle ( );
  operator IPresSpaceHandle ( );
}
```

New Interface in IWindow for Event Dispatching & Associated Changes to IMouseHandler

Our enhanced mouse handler now supports mouse move messages, whereas the original Motif implementation did not. This handler does not work on Motif today because none of the Widgets request move messages(limited exceptions to this rule, like sash widget in ISplitCanvas). We cannot use our current handler philosophy on Motif, which basically states that we register all callbacks that a handler may be interested in seeing.

Motif is designed to be a server client and is set up to work over a network. This design forces Motif applications to try to cut down on network traffic by asking an application to register only the necessary events on a per widget basis. We have to be selective in what events we request especially with an event that occurs as often as a mouse move. We have enhanced the current interface to specify which controls (widgets) we want to see mouse move events for.

New interface on IWindow for Event Dispatching

The following functions to allow event handlers for a window to receive or not receive certain messages have been added:

- IWindow::startHandling (const EventMask& mask)
- IWindow::stopHandling (const EventMask& mask)
- IWindow::isHandling (const EventMask& mask) const

where EventMask is a nested class that derives from IBitFlag. The public static event masks that IWindow provides are:

- IWindow::EventMask
- someMouseMoves,
- allMouseMoves,
- mouseEntersLeaves;

On OS/2 and Windows, someMouseMoves and allMouseMoves are equivalent. Specifying mouseEntersLeaves causes someMouseMoves to be enabled as well on some platforms (Windows NT). Calling stopHandling prevents any handlers from being called for that window to process the specified type of event. Conversely, calling startHandling allows handlers to be called for the event.

Calling stopHandling does not prevent IWindow::defaultProcedure from being called. No attempt is made, however, to call private default handlers of a window. Therefore, turning off receipt of messages can change default behavior of a window. For example, it can prevent the user from being able to move split bar of an ISplitCanvas on OS/2 or Windows.

Reference counting to allow multiple calls to startHandling and stopHandling for an event type is used. A default bool argument to stopHandling to force off receipt of messages to handlers, regardless of the reference counting state has been implemented. By default, handlers of all windows will receive all events on OS/2 and Windows. For performance reasons, Motif handlers will not receive mouse move or mouse enter/leave events by default. Note that generally an application should not need to call these functions. To cover the most common case where these functions need to be called, we are enhancing IMouseHandler to call them on the application's behalf.

Cases where an application will need to call IWindow::startHandling or stopHandling include:

- The application is processing mouse move events in a handler that does not derive from IMouseHandler.
- The application prefers to not modify or cannot modify the call to the IMouseHandler constructor. For example, it uses a mouse handler class provided by a third party.
- The application needs to dynamically start and stop the receipt of mouse messages, which IMouseHandler does not support.

Style Added to IMouseHandler

An optional style argument to the IMouseHandler constructor has been added. These styles dictate the type of mouse messages that the handlers of the window can receive. As a result, they actually affect more than just the single mouse handler. The constructor now looks like:

```
MouseHandler ( const Style& style = defaultStyle() );
```

where Style is a nested class that derives from IBitFlag. The public static styles that IMouseHandler provides are:

- static const IMouseHandler::Style
- noMouseMoves,
- someMouseMoves,
- allMouseMoves,
- mouseEntersLeaves;

The default style is someMouseMoves. This supports backward compatibility with existing mouse handlers written for OS/2 and Windows by requesting a portion of the mouse move events for the window's handlers. On OS/2 and Windows, someMouseMoves and allMouseMoves are equivalent. The IMouseHandler constructor calls IWindow::startHandling or stopHandling as appropriate (these styles map to the EventMask flags). If the application constructs the handler with the noMouseMoves style, then code to be added to IMouseHandler::dispatchHandlerEvent prevents its mouseMoved virtual function from being called even if the handler does receive a mouse move event. The same rule applies to not specifying the mouseEnters or mouseLeaves style. In this case the new mouseEnter and mouseLeave virtual functions described at the bottom of this document are never called.

The following standard auxiliary style members are added to IMouseHandler:

- static Style
- IMouseHandler::defaultStyle();
- static void
- IMouseHandler::setDefaultStyle (const Style& style);
- static const Style
- classDefaultStyle;

Deprecated IMouseHandler::mousePointerChange

In order to increase performance on Motif with processing mouse messages, this product has deprecated IMouseHandler::mousePointerChange. This functionality can be better handled by calling IFrameWindow::setMousePointer. To be able to change the mouse pointer dynamically based on what portion of the window the mouse is over, override IMouseHandler::mouseMoved to call IWindow::setMousePointer (mouseMoved is the best place for determining that the mouse changed and its location). You could get inconsistent behavior if you use both IWindow::setMousePointer and an IMouseHandler to change the mouse pointer for a window, depending on the order that you call setMousePointer and attach the handler (and whether the handler returns true or false).The product does not implement this virtual function on Motif.

Pointer Enter/Leave Support

The following protected virtual functions in IMouseHandler are new:

```
virtual bool
IMouseHandler::mouseEnter ( IEvent& event ),
IMouseHandler::mouseLeave ( IEvent& event );
```

IMouseHandler calls these virtual functions when the mouse pointer crosses the dispatching window's boundary.

The return value from processing the mouseEnter and mouseLeave virtual function has no effect on whether the mouseMoved virtual function gets called. The same is true the other way around as well. IMouseHandler does not support a time delay (your application can use ITimer to achieve this support).

MakeIMousePointerEventAvailable on OS/2 and Windows Only

This class is not implemented on Motif, but it is not deprecated either. In the class library it is only used by the deprecated IMouseHandler::mousePointerChange function and the IC_PMWIN-only IMousePointer class.

IMouseConnectionTo Deprecated

This class template has been deprecated and it is not implemented on Motif. Replace all usage of IMouseConnectionTo with one of the following two classes, as appropriate:

IMouseClickConnectionTo
IMouseMoveConnectionTo

Toolbar on Motif

This release of the Open Class Library now contains IToolBar, IToolBarButton and the other toolbar related classes with limited functionality.The following Motif toolbar functionality is provided at this time:

- The ability to create a toolbar at any location position, above, below, left and right of the client or as a floating toolbar.
- Support for multiple toolbar in same location.
- Addition of any window to the toolbar.
- ToolBar buttons without the custom button draw override capability.
- Bitmap view and text view for buttons.
- IToolBar::setLocation() to move a toolbar from one location to another.

Due to Motif system restrictions, certain toolbar movements are restricted. (A new function IToolBar::isMoveValid() has been created to verify toolbar movements. See the Open Class Reference for details on this function as well at toolbar movement restrictions.)

- SetBitmap(), setText() of buttons.
- Toolbar filtering.
- Toolbar container and its functions.

The following Motif toolbar functionality has not been implemented at this time, but is planned for the next release:

- The Open Class supplied bitmaps are not provided (at this time all bitmaps must be user supplied.

- IToolBar::bitmapAndText view shows only text.
- Custom drawing through IToolbarButton and ICustomButton virtual functions.
- ICustomButtonDrawEvent is not implemented yet.
- Button latching and disabled buttons not implemented yet.
- Drag/drop of toolbar or buttons not implemented yet.

The following sections define deprecated and ignored functions and explain how to use these functions in the Open Class Library. To develop portable applications, you should be aware of these conventions.

Deprecated Functions

As the Open Class Library functionality increases, there are situations where we must change the interface to improve the quality and design. We identify the interface that is deprecated and provide this information so you can migrate to replacement classes and functions.

There are several important guidelines regarding deprecated functions:

- Usually the implementation of an deprecated function calls the function that has replaced it.
- Typically, we remove the deprecated interface in a version of library in the next major release of the library. We do not document deprecated interface in the main body of the reference manual. Instead, it is documented in a section which identifies deprecated interface and replacement classes and functions if they are available.

The following are changes in the language/methods/enums/types in this release of the IBM Open Class Library:

Type	Name	Replaced by
class	IBase	
class	IBase::Version	
function	IBase::asString() const	IStringGenerator::stringFor() const (where the IStringGenerator was constructed passing in an instance of an IStringGeneratorasString generator function)
class	IBasicColor	IBaseColor
class	IGArc	IGCurve2D
class	IGBitmap	IGImage, IImage
class	IGEllipse	IGEllipse2D, IEllipse
class	IGLine	IGLine2D, ILine
class	IGList	IGraphicGroup
class	IGPie	IGPie2D, ICurve
class	IGPolyline	IGPolyline2D, IGPolyline
class	IGraphic	IMGraphic
class	IGraphicBundle	IGrafBundle
class	IGraphicContext	IGrafPort
class	IGRectangle	IGRect2D, IPolygon
class	IGRegion	IGArea, IArea

class	IGString	IGraphicText
class	IRootGrafPort	IExtendedRootGrafPort
class	ITransformMatrix	
class	IWindow::BidiSettings	IBidiSettings
class	IMouseConnectionTo	IMouseClickConnectionTo and IMouseMoveConnectionTo
typedef	Boolean	bool
typedef	IBaseErrorInfo::IErrorInfo	IBaseErrorInfo
typedef	IColor::SysColor	IColor::SystemColor
typedef	IContextHandle	IAnchorBlockHandle
typedef	ICoordinateSystem::Orientation	ICoordinateSystem::EOrientation
enum	IColor::Color	IBaseColor::EPredefinedColor
enum	IColor::SystemColor	IColor::ESystemColor
enum	EDeviceCoordinateSystem	ICoordinateSystem::EOrientation
enum	IImage::EDitherType	IGImagePixelAccessor::EDitherType
enum	IImage::EImageFormat	IGImagePixelAccessor::EImageFormat
enum	INonGUIApplication::PriorityClass	INonGUIApplication::EProcessPriority
enum	IWindow::BidiLayout	IBidiSettings::BidiLayout
enum	IWindow::BidiNumeralType	IBidiSettings::BidiNumeralType
enum	IWindow::BidiTextOrientation	IBidiSettings::BidiTextOrientation
enum	IWindow::BidiTextShape	IBidiSettings::BidiTextShape
enum	IWindow::BidiTextType	IBidiSettings::BidiTextType
constructor	IBidiSettings (const IGraphicContext&)	IBidiSettings(const IGrafPort&)
constructor	IBitFlag (unsigned long)	IBitFlag(unsigned long, const unsigned long[])
constructor	IBitFlag (unsigned long, unsigned long)	IBitFlag(unsigned long, const unsigned long[])
constructor	IColor(Color color)	IColor(IColor::EPredefinedColor)
constructor	IColor(SystemColor value)	IColor(IColor::ESystemColor)
constructor	IColor(unsigned long pixel)	IColor(long index)
constructor	IImage(IPresSpaceHandle psh, int , int)	
constructor	IImage(const IString& .....)	IGImagePixelAccessor
data	IEntryField::lowerCase	Create a handler to convert characters to lowercase
data	IEntryField::upperCase	Create a handler to convert characters to uppercase
data	IKeyboardEvent::ulAltFlag	
data	IKeyboardEvent::ulAltMask	
data	IKeyboardEvent::ulCharacterFlag	
data	IKeyboardEvent::ulCompositeFlag	
data	IKeyboardEvent::ulCtrlFlag	
data	IKeyboardEvent::ulCtrlMask	
data	IKeyboardEvent::ulForCompositeFlag	
data	IKeyboardEvent::ulInvalidCompositeFlag	
data	IKeyboardEvent::ulRepeatedFlag	

data	IKeyboardEvent::ulScanCodeFlag	
data	IKeyboardEvent::ulShiftFlag	
data	IKeyboardEvent::ulShiftMask	
data	IKeyboardEvent::ulUncombinedFlag	
data	IKeyboardEvent::ulUpTransitionFlag	
data	IKeyboardEvent::ulVirtualFlag	
constructor	INonGUIThread(const IReference<IThreadFn>&)	INonGUIThread(IThreadFn*)
function	IBidiSettings::apply(const IGraphicContext&)	apply(const IGrafPort&)
function	IBitFlag::asExtendedUnsignedLong ( ) const	No true replacement. Classes using IBitFlag objects must provide their own mapping logic (IWindow::convertToGUIStyle is an example).
function	IBitFlag::asUnsignedLong ( ) const	No true replacement. Classes using IBitFlag objects must provide their own mapping logic (IWindow::convertToGUIStyle is an example)
function	IBitFlag::setValue(unsigned long, unsigned long)	IBitFlag::setValue(const IBitFlag&)
function	ICanvas::defaultPushButton ( ) const	IWindow::defaultEmphasisButton
function	ICanvas::origDefaultButtonHandle ( ) const	IWindow::defaultPushButton
function	IClipboard::setOwner	Specify the owner when constructing the IClipboard (construct a new IClipboard object if necessary)
function	IColor::asPixel() const	IColor::index()
function	IColor::systemColor() const	IColor::runtimeGuiColor()
function	IColor::value() const	IColor::predefinedColor
function	ICurrentThread::appContext	ICurrentThread::anchorBlock
function	ICurrentThread::appShell	IThread::applicationShell or IWindow::desktopWindow()->handle()
function	ICustomButtonDrawEvent::graphicContext	ICustomButtonDrawEvent::grafPort
function	IDrawingCanvas::graphicContext	IDrawingCanvas::grafPort
function	IDrawingCanvas::graphicList	IDrawingCanvas::graphicGroup
function	IDrawingCanvas::setGraphicContext	IDrawingCanvas::setGrafPort
function	IDrawingCanvas::setGraphicList	IDrawingCanvas::setGraphicGroup
function	IEntryField::hasChanged	IEntryField::hasTextChanged
function	IEntryField::setChangedFlag	IEntryField::setTextChangedFlag
function	IImage::loadFromFile	IGImagePixelAccessor::loadFromFile
function	IImage::writeToFile	IGImagePixelAccessor::writeToFile
function	IInfoArea::handleEventsFor	IInfoArea::startShowingMenuInfoFor
function	IInfoArea::inactiveText	disabledText
function	IInfoArea::setInactiveText	setDisabledText
function	IInfoArea::stopHandlingEventsFor	IInfoArea::stopShowingMenuInfoFor
function	IMouseHandler::mousePointerChange	IMouseHandler::mouseEnter and mouseLeave (or mouseMoved), calling IFrameWindow::setMousePointer
function	IMultiLineEdit::setChangedFlag	IMultiLineEdit::setTextChangedFlag
function	IMultiLineEdit::resetChangedFlag	IMultiLineEdit::resetTextChangedFlag
function	IMultiLineEdit::isChanged	IMultiLineEdit::hasTextChanged

function	INonGUIApplication::adjustPriority	INonGUIApplication::setProcessPriority
function	INonGUIApplication::setPriority	INonGUIApplication::setProcessPriority
function	INonGUIThread::priorityLevel	INonGUIThread::threadPriority
function	INonGUIThread::setPriority	INonGUIThread::setThreadPriority
function	INonGUIThread::	
function	INotificationEvent::setNotifierAttrChanged(booltrue)	
function	INotificationEvent::setEventData(const IEventData&)	
function	INotificationEvent::setObserverData(const IEventData&)	
function	INotificationEvent::hasNotifierAttrChanged	
function	INotificationEvent::eventData	
function	INotificationEvent::observerData	
function	INotificationEvent::notifier	
function	INotifier::addObserver(IObserver&, const IEventData&)	
function	IPaintEvent::setGraphicContext	IPaintEvent::setGrafPort
function	IResourceLibrary::loadAccelTable(unsigned long, const IWindowHandle&)	IResourceLibrary::loadAccelTable(unsigned long)
function	ISetCanvas::setText	ICanvas::setBorderText
function	ISetCanvas::text	ICanvas::borderText
function	IStandardNotifier::addObserver(IObserver&, const IEventData&)	
function	IWindow::addObserver(IObserver&, const IEventData&)	
function	IWindow::applyBidiSettings	IBidiSettings::apply
function	IWindow::enableFastWindowWithHandle(bool enable = true)	IWindow::reserveUserWindowWord
function	IWindow::isBidiSupported	IBidiSettings::isBidiSupported
function	IWindow::isFastWindowWithHandleEnabled()	IWindow::isUserWindowWordReserved
constant	True	true
constant	False	false
constant	ICommand::kCopyToId	kSaveAsId
constant	ICoordinateSystem::originLowerLeft	ICoordinateSystem::kOriginLowerLeft
constant	ICoordinateSystem::originUpperLeft	ICoordinateSystem::kOriginUpperLeft
constant	IEntryField::lowerCase	
constant	IEntryField::upperCase	
constant	IHelpWindow::using	usingHelp
constant	IKeyboardEvent::ulAltFlag	
constant	IKeyboardEvent::ulAltMask	isAltDown()
constant	IKeyboardEvent::ulCharacterFlag	
constant	IKeyboardEvent::ulCompositeFlag	
constant	IKeyboardEvent::ulCtrlFlag	
constant	IKeyboardEvent::ulCtrlMask	isCtrlDown()

constant	IKeyboardEvent::ulForCompositeFlag	
constant	IKeyboardEvent::ulInvalidCompositeFlag	
constant	IKeyboardEvent::ulRepeatFlag	
constant	IKeyboardEvent::ulScanCodeFlag	
constant	IKeyboardEvent::ulShiftFlag	
constant	IKeyboardEvent::ulShiftMask	isShiftDown()
constant	IKeyboardEvent::ulUncombinedFlag	
constant	IKeyboardEvent::ulUpTransitionFlag	
constant	IKeyboardEvent::ulVirtualFlag	
constant	IProgressIndicator::armTickOffsetId	armChangeId
constant	IProgressIndicator::armPixelOffsetId	armChangeId
constant	IScrollBar::autoSize	
constant	IScrollBar::scrollBoxPositionId	scrollBoxChangeId, scrollBoxTrackId
constant	IString::and	bitAnd
constant	IString::or	bitOr
constant	IString::exclusiveOr	bitExclusiveOr
constant	IToolBar::hidden	
constant	IToolBar::noDragDrop	
constant	IToolBarButton::noDragDelete	
constant	kLeftHand	ICoordinateSystem::kOriginUpperLeft
constant	kRightHand	ICoordinateSystem::kOriginLowerLeft

Standard Library Overview

The Standard C++ Library and the IBM Open Class Library (IOC) are not entirely compatible. Choose between them if you are going to use either.

Continue using the IOC if you are adding to code that already uses the IOC extensively. Otherwise, consider the strengths and weaknesses of each library as a whole by reading the online documentation for each.

The table below lists the categories of classes that the libraries have in common if there are differences that could affect your choice of libraries. The IOC also includes categories of classes such as multimedia and 2-D graphics, which are not found at all in the Standard C++ Library.

Common Category of Classes	**Considerations When Using IBM Open Class Library (IOC)**	**Considerations When Using Standard C++ Library**
Iostream		Supports wchar streams Locale sensitivity built-in
Numerics	Supports complex only	Supports linear algebra and complex

Strings	Whether to use IText or IString IText versus IString IText has no numeric support IText is tied to the IOC Internationalization and 2-D Graphics frameworks IText has styles (e.g. highlighting, italics) IString uses styles via changing fonts IString is used by User Interface Classes	Supports wide character strings
Collections	Uses collection classes Richer functionality through more abstract data types More complicated to use than the Standard C++ Library Uses object-oriented style	Uses Standard Template Library Higher performance, smaller code generation Uses generic programming style
Localization	Uses Internationalization Framework Uses locales and Unicode Uses Transcoding classes	Uses Standard C++ Library localization Used by I/O Stream classes

The Standard C++ Library is part of the international standard for C++ and is included with VisualAge C++, V 4.0.

The Standard C++ Library provides functionality that strives for compiler independence, usefulness, and efficiency. The library includes a set of container classes and algorithms referred to as the Standard Template Library (STL).

Stroustrup writes about the standard library, including its design constraints, from chapter 16 forward to 22. The other major and more detailed source provided with this compiler is the Dinkum C++ Library Reference.

Chapter 2. Collection Classes

Introduction to the Collection Classes

Collections are used to store and manage elements (or objects) of a user-defined type. Different collections have different internal structures, performance characteristics, and access methods for storage and retrieval of objects. The Collection Classes implement the common collection types, such as trees, relations and ordered lists. They provide a framework of properties to help you decide which abstract collection type is appropriate in a given situation, and allow you to choose how the abstract data type you have chosen is implemented.

The Collection Classes let you choose the appropriate collection type for a given situation by providing collection classes which have a systematic and consistent combination of basic properties. These properties help you to select the appropriate level of abstraction. For example, you may have the choice between using a bag and a key sorted set. The properties of these two collections will help you decide which one is more appropriate.

The Collection Classes offer you a choice of implementations for each type of collection. Each abstract collection class has a common interface with all of its possible implementations. It is easy to replace one implementation with another for performance reasons or if the requirements of your application change.

Types of Classes

The Collection Classes are divided into three types:

- Trees—recursive collections of nodes, where each node holds an element and has a given number of nodes as children
- Flat collections—the most common types of collections, include abstractions such as sequence, set, bag, and map. Unlike trees, flat collections have no hierarchy of elements or recursive structure.
- Auxiliary classes—support other classes and include classes for cursors, iterators, and simple and managed pointers.

Hierarchy and Design of the Collection Classes

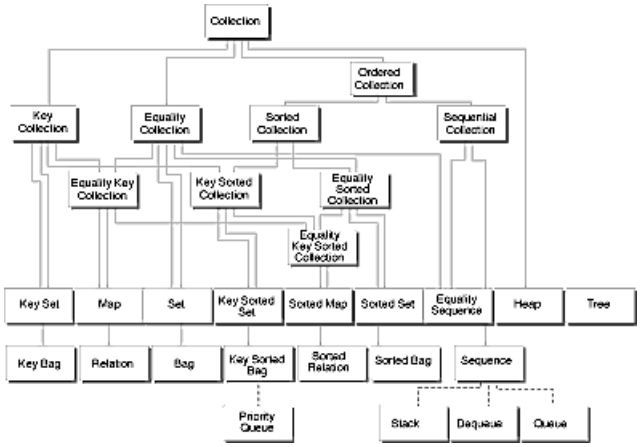
This file contains the following subjects:

- Collection Class Hierarchy
- Overall Implementation Structure
- Element Functions and Key-Type Functions
- Collection Class Polymorphism

Collection Class Hierarchy

The classes in the Collection Classes are all related through the hierarchy of abstract classes shown in the figure below:

© Copyright IBM Corp. 1998



In the figure, abstract collections have a gray shadow, concrete collections have a black shadow, and restricted-access collections have a white shadow. The leaves of the collection hierarchy (those collections that have no derived classes within the hierarchy tree) define the collection for which concrete implementations are provided. The lines in the figure represent an 'is-a' relationship from a lower collection to the collection above it. For example, a set is an equality collection, which is a collection.

Dotted lines show a 'based-on' relationship, not an actual derivation.

Overall Implementation Structure

The abstract collection classes represent a concept, and classes derived from it represent implementations of the concept. With abstract classes, you can program to a more generalized interface without knowing what specific collection type your code will operate on. Implementation details can be left for later. For example, when working with a set, you can write your program to use the interfaces of the abstract classes IASet or IAEqualityCollection, rather than the concrete classes ISet, IGSet, ISetAsBstTree, and so on.

The names of the abstract collection classes start with the letters IA.

Each abstract collection type has several possible implementations. Some of these implementations are basic; that is, the collection class is implemented directly as a concrete class. These basic implementations include:

- AVL trees
- Hash tables
- Linked sequences
- Tabular sequences

Variant implementations of the same collection behave externally in the same way but may offer improved performance for a particular application, depending on the collection’s characteristics. Sets can be implemented, for example, as AVL trees, lists, or hash tables.

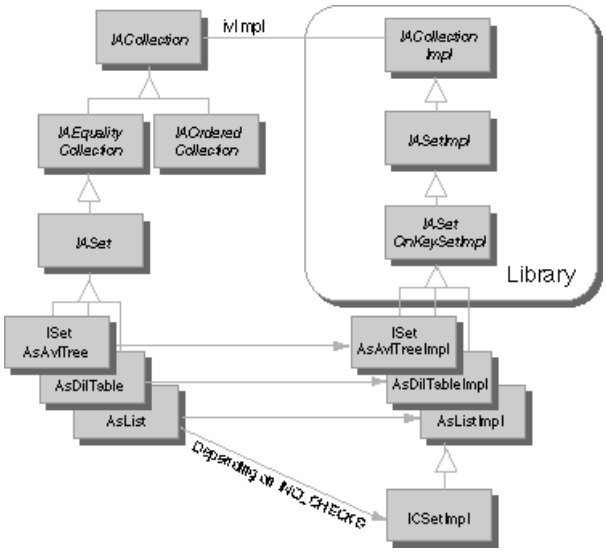
Default implementations are provided for every collection. Two default classes are provided for each abstract data type: a class that is instantiated only with the element type (and possibly the key type) and one that is instantiated by passing in element-specific functions. In many cases, you do not need to concern yourself with

the choice of implementation. If you choose not to specify one, the Collection Classes will use a reasonable default implementation class.

Each class falls within one of the following categories: concrete, typed implementation, typeless implementation, and abstract. The figure below illustrates the relationships (indicated by arrows) between the categories of classes for the set collection classes. Arrows indicate a relationship between classes:

- Instantiates (arrow with dashed line)
- Is-a (line with triangle)
- Has a (arrow with solid line)

Overall Structure of Collection Classes



The class ISet uses an AVL tree as the default implementation. The other implementation variants are linked list and diluted table. The three implementation variants ISetAsAvlTree, ISetAsList, and ISetAsDilTable are subclasses of IASet. If you do not want to deal with implementation variants, you can just use the default class ISet.

The Based-On Concept

The Collection Classes achieve a high degree of implementation flexibility by basing several collection class implementations on other abstract classes, rather than by implementing them directly through a concrete implementation variant of the class. This design feature results in an implementation path rather than the selection of an implementation in a single step. The Collection Classes contain type definitions for the most common implementation paths.

The element functions that are needed by a particular implementation depend on all collection class templates that participate in the implementation. While ISet requires at least element equality to be defined, an AVL tree implementation of this set also requires the element type to provide a comparison function. A hash table implementation also requires the element type to have a hash function. The required element functions for all predefined implementation variants are listed for individual collection types.

For a concrete implementation, such as a set based on a key-sorted set that is in turn based on a tabular sequence, these class templates are plugged together so that the elements only need to define the operations that are needed for the specific type of collection being used.

Element Functions and Key-Type Functions

The member functions of the Collection Class Library call other functions to manipulate elements and keys. These functions are called element functions and key-type functions, respectively. Member functions of the Collection Class Library may, for example, use the element's assignment or copy constructors for adding an element, or they may use the element's equality operator for locating an element in the collection.

Collection Class Polymorphism

Polymorphic use of collections differs from polymorphism of the element type. Polymorphic use of collections means that a function can specify an abstract collection type for its argument, for example IACollection, and then accept any concrete collection given as its actual argument. Element polymorphism means that you can use the collections with any elements that provide basic operations like assignment and equality. This section deals with the polymorphic use of collections rather than elements.

Each abstract class is defined by its functions and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. Elements can be added to a collection, and a collection can be iterated over. A polymorphic function on collections that uses only properties of the most abstract view might be to print all elements.

Collections with more specialized element properties, such as equality or key equality, also provide functions for retrieving element occurrences by a given element or key value. Ordered collections provide the notion of a well-defined ordering of element occurrences, either by an element ordering relation or by explicit positioning of elements within a sequence. Ordered collections define operations for positional element access. Sorted collections provide no further functions, but define a more specific behavior, namely that the elements or their keys are sorted.

The properties represented by abstract collection classes are combined through multiple inheritance. For example, the abstract collection class IAEqualitySortedCollection, for example, combines the properties of element equality and element sorting, which implies being ordered. If a polymorphic function uses IAEqualitySortedCollection as its argument type, the argument will be sorted, and the function can use functions such as contains that are only defined for collections with element equality.

Collection Characteristics

Four basic properties are used to differentiate between different flat collections:

- Ordering—Whether a next or previous relationship exists between elements.

- Access by key—Whether a part of the element (a key) is relevant for accessing an element in the collection. When keys are used, they are compared using relational operators.
- Equality for elements—Whether equality is defined for the element.
- Uniqueness of entries—Whether any given element or key is unique, or whether multiple occurrences of the same element or key are allowed.

Ordering of Collection Elements

The elements of a flat collection class can be ordered in three ways:

- Unordered collections have elements that are not ordered.
- Sorted collections have their elements sorted by an ordering relation defined for the element type. For example, integers can be sorted in ascending order, and strings can be ordered alphabetically. The ordering relation is determined by the instantiations for the collection class. For elements where the ordering relation returns the same position, elements are added in chronological order.
- Sequential collections have their ordering determined by an explicit qualifier to the add function, for example, addAtPosition.

A particular element in a sorted collection can be accessed quickly by using the ordering relation to determine its position. Unordered collections can also be implemented to allow fast access to the elements, by using, for example, a hash table or a sorted representation. The Collection Class Library provides a fast locate function that uses this structure for unordered and sorted collections. Even though unordered collections are often implemented by sorting the elements, do not assume that all unordered collections are implemented in this way. If your program requires this assumption to be true, use a sorted collection instead.

For each flat collection, the Collection Class Library provides both unordered and sorted abstractions. For example, the Collection Class Library supports both a set and a sorted set. The ordering property is independent of the other properties of flat collections. You can make a given flat collection unordered or sorted regardless of its other properties.

Access by Key

A given flat collection can have a key defined for its elements. A key is usually a data member of the element, but it can also be calculated from the data members of the element by some arbitrary function. Keys let you:

- Organize the elements in a collection
- Access a particular element in a collection

For collections that have a key defined, an equality relation must be defined for the key type. Thus, a collection with a key is said to have key equality.

Equality Relation

The Collection Class Library provides sorted and unsorted versions of maps and relations, for which both key and element equality must be defined. These collections are similar to key set and key bag, except that they define functions based on element equality, namely union and intersection. The add function behaves differently for maps and relations than it does for key set and key bag collections.

A flat collection can have an equality relation defined for its elements. The default equality relation is based on the element as a whole, not just on one or more of its data members (for example, the key). For two elements to be equal, all data members of both elements must be equal. The equality relation is needed for functions such as those that locate or remove a given element. A flat collection that has an equality relation has element equality.

Uniqueness of Entries

The terms unique and multiple relate to the key, in the case of collections with a key. For collections with no key, unique and multiple relate to the element.

In some flat collections, such as map, key set, and set, no two elements are equal or have equal keys. Such collections are called unique collections. Other collections, including relation, key bag, bag, and heap, can have two equal elements or elements with equal keys. Such collections are called multiple collections.

For those multiple collections that have keys with element equality (relation and sorted relation), elements are always unique while keys can occur multiple times. In other words, if element equality is defined for a multiple collection with key, element equality is tested before inserting a new element.

A unique collection with no keys and no element equality is not provided because a containment function cannot be defined for such a collection. A containment function determines whether a collection contains a given element.

The behavior during element insertion (when one of the add... methods is applied to a collection) distinguishes unique and multiple collections. In unique collections, the add function does not add an element that is equal to an element that is already in the collection. In multiple collections, the add function adds elements regardless of whether they are equal to any existing elements or not.

Iteration

The Collection Classes give you two methods of iteration:

- Using cursors
- Using the allElementsDo function together with applicators or applicator functions

Ordered (including sorted) collections have a well-defined ordering of their elements, while unordered collections have no defined order in which the elements are visited in an iteration. However, in both cases, each element is visited exactly once.

You cannot add or remove elements from a collection while you are iterating over a collection, or all elements may not be visited once. You cannot use any of the iterations described in this section if you want to remove all of the elements of a collection that have a certain property. Use the function removeAll that takes a predicate function as an argument.

Iteration with Cursors

To use the collection classes, you also need a cursor class for referencing an element in a collection, and an applicator class or function for iterating over a collection.

Cursors and iterators give you convenient methods for accessing the elements stored in the collections.

- cursors
- tree cursors
- applicators
- constant applicators

A cursor is a reference to the position of an element in a collection. If the position of the element changes, the cursor is invalidated. A cursor is always associated with a collection. The collection is specified when the cursor is created. Each collection function that takes a cursor argument has a precondition that the cursor actually belongs to the collection. Simple functions, such as advancing the cursor, are also functions of the cursor itself.

With cursors the Collection Classes provide:

- An iteration scheme that is simpler than using iterators.
- The ability to define functions that return cursors. Such functions can give you fast access to an element if it exists, or indicate the non-existence of an element by returning an invalid cursor.

Cursors are only temporarily defined. As soon as elements are added to or removed from the collection, existing cursors become undefined. One of the three following situations occurs:

1. The cursor is invalidated (isValid will return false).
2. The cursor remains valid and points to an element of the collection; however, it may point to a different element than before.
3. The cursor remains valid but no longer points to an element of the collection.

Do not use an undefined cursor as an argument to a function that requires the cursor to point to an element of the collection.

Each concrete collection class, such as ISet<int>, has an inner definition of a class Cursor that can be accessed as ISet<int>::Cursor.

Because abstract classes declare functions on cursors just as concrete classes do, there is a base class ICursor for these specific cursor classes. To allow for the creation of specific cursors for all kinds of collections, every abstract class has a virtual member function newCursor, which creates an appropriate cursor for the given collection object.

Iteration with allElementsDo

Cursor iteration has two possible drawbacks:

- For unordered collections, the explicit notion of an (arbitrary) ordering may be undesirable for stylistic reasons. For example, it could mislead you (or another programmer) into perceiving or exploiting an order where in fact the order does not exist or is not guaranteed.
- Iteration in an arbitrary order might be done more efficiently using something other than cursors. For example, with tree representations, a recursive descent iteration may be faster than the cursor navigation, even though the time for extra function calls must be considered.

The Collection Classes provide the allElementsDo function that addresses both drawbacks by calling a function that is applied to all elements. The function returns an IBoolean value that tells whether the iteration should be continued or not. For ordered collections, the function is applied in this order. Otherwise the order is unspecified.

The function that is applied in each iteration step can be given in two ways: directly as a C++ function, or by defining the function as a method of a user-defined applicator class:

- As a C++ function: Code the function that you want to be applied to all elements as a C++ function, then use allElementsDo to apply the function to the elements.
- As an object of an applicator class: Code the function as a member function of an applicator class that you create (for example, MyApplicatorClass) and let the applicator apply this function to every element, by using allElementsDo(myApplicatorObject), where myApplicatorObject is an object of MyApplicatorClass.

Flat Collections

The following flat collections are provided by the Collection Class Library:

- sets—key sets, sorted sets, key sorted sets
- bags—key bags, sorted bags, key sorted bags
- queues—priority queues
- deques
- heaps
- maps—sorted maps, relations, sorted relations
- sequences and equality sequences
- stacks

Combinations of Properties for Flat Collections

The figure below shows the flat collection that results from each combination of properties. For example, Map appears in the Unique, Unordered column for the Key, Element Equality row. This means that a map is unordered, each element is unique, keys are defined, and element equality is defined. This implies that there are no flat collections that have all of the following properties:

- The collection is ordered
- The collection is sequential
- The collection allows an element to appear more than once

- Keys are defined for elements in the collection

The rationale for not implementing collections with these combinations of properties is that there is no reason to choose them over another collection that is already available. For example, for an ordered collection that is sequential and offers access by key, the key access would only have advantages if the elements are stored in a position depending on their key. Because they are not, there is no flat collection with key access that maintains a sequential order.

		Unordered		Ordered		
				Sorted		Sequential
		Unique	Multiple	Unique	Multiple	Multiple
Key (key equality must be defined)	**Element Equality**	Map	Relation	Sorted map	Sorted relation	N/A
	No Element Equality	Key set	Key bag	Key sorted set	Key sorted bag	N/A
No Key	**Element Equality**	Set	Bag	Sorted set	Sorted bag	Equality sequence
	No Element Equality	N/A	Heap	N/A	N/A	Sequence

For example, the default set collection:

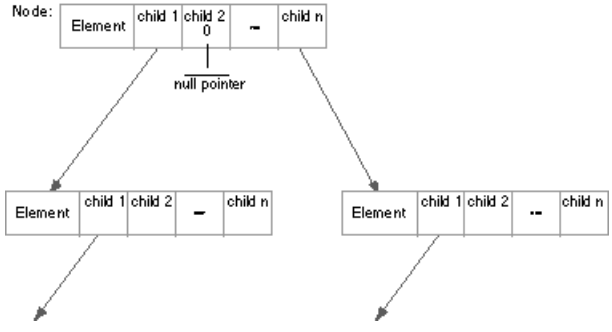
- does not allow an item to be added if it is already present in the collection, ensuring that elements are unique
- is not sorted
- cannot locate elements by key

Trees

Trees can be described either as structures whose elements have a hierarchy or as a special form of recursive structure. Recursively a tree can be described as a node (parent) with pointers to other nodes (children). Every node has a fixed number of pointers, which are set to null at initialization time. Insertion of a new node involves setting a pointer in the parent so that it points to the inserted child. The figure below illustrates the structure of an n-ary tree.

One node is the entry point to the tree. This node is designated as the root. Nodes without any pointers to other nodes are called leaf nodes or terminal nodes.

The Structure of N-ary Trees



Similarly, you can obtain tree-like or recursive structures by implementing the array of children of a node as a flat collection of nodes. This will give you different functionality for the children, for example, the ability to locate a child with a given value.

Trees in general are more useful for searching elements than for adding and deleting elements. For this reason, they are often called search trees. The descriptions of AVL and B* trees explain why trees are well-suited for searching. Generally, you can locate and insert elements in collections implemented as trees faster than you can in collections implemented as lists. However, if you only want to iterate through elements in a collection, it is faster to iterate through the elements of a list.

Backward-Compatible Items

The following are items of former releases that are compatible with the new release:

Reference Classes

Within the new release, reference classes are no longer necessary for polymorphic use of the collections. The concrete collection classes are now directly derived from the abstract class hierarchy. A linkage of abstract and concrete classes through reference classes is therefore superfluous. Nevertheless you can continue using the reference class syntax in existing programs.

IIterator and IConstantIterator

The classes IIterator and IConstantIterator are now called IApplicator and IConstantApplicator. The new names express more precisely what the purpose of objects from these classes is: They do not iterate over a collection themselves but they provide a function that is applied to the elements of a collection during iteration with allElementsDo.

The classes IIterator and IConstantIterator are still available but not recommended.

The forCursor macro

Instead of the forCursor macro the forICursor macro is introduced. The forCursor macro is still available but - as with the Iterator classes - you should prefer using the new version.

IECOps

Up to now all implementation variants of the collections bag, set, sorted bag and sorted set used the element operation class IECOps. In the new release, these collections require only class ICOps which is a subset of IECOps. That means, in the new release class IECOps is no longer needed, yet it is still available.

Naming Conventions

New names have been introduced for the implementation variants as well as for the corresponding header files. The old names can still be used in existing programs. Consider the key set as example:

Old Names		New Names	
IKeySet	ikeyset.h	IKeySet	iks.h
		IKeySetAsAvlTree	iksavl.h
IKeySetOnBSTKeySortedSet	iksbst.h	IKeySetAsBstTree	iksbst.h
IHashKeySet	ihshks.h	IKeySetAsHshTable	ikshsh.h
IKeySetOnSortedLinkedSequence	ikssls.h	IKeySetAsList	ikslst.h
IKeySetOnSortedTabularSequence	iksstsh.h	IKeySetAsTable	ikstab.h
IKeySetOnSortedDilutedSequence	ikssds.h	IKeySetAsDilTable	iksdil.h

Incompatibilities

The following items are not compatible with the new collection class library release:

New class hierarchy

The abstract hierarchy makes use of virtual inheritance. When you subclass from a Collection Class and implement your own copy constructor, you must initialize the virtual base class IACollection<Element> in your derived classes.

The newCursor method

Different from previous releases, the return type of the newCursor method is now for any collection a pointer to the abstract cursor class ICursor (ICursor*).

Deriving from Reference Classes

You can still derive from reference classes without overriding existing collection class member functions. Yet, you can no longer override existing collection class functions and use your derived collection class in a polymorphic way without additional effort.

Collection Class Library Exceptions

This file contains information about the following subjects:

- Exceptions Defined and Used by the Collection Classes
- Exception Causes
- Exceptions Caused by System Failures and Restrictions

Exceptions Defined and Used by the Collection Classes

The exceptions defined and used in the Collection Class Library are shown in the table below (classes defined by the Collection Class Library are shown in **bold**). They all derive from IException, directly or indirectly.

Hierarchy of Exceptions Defined and Used by the Collection Classes

```
IException
   ICollectionResourceException
   ICollectionLockException
   ICollectionUnlockException
   ICollectionLockTimeOutException*
```

```
IPreconditionViolation      IChildAlreadyExistsException
   ICursorInvalidException
   IEmptyException
   IFullException
   IIdenticalCollectionException
   InvalidReplacementException
   IKeyAlreadyExistsException
   INotBoundedException
   INotContainsKeyException
   IPositionInvalidException
   IRootAlreadyExistsException   IResourceExhausted
   IOutOfMemory
```

*not on AIX .

Exception Causes

A *precondition* of a called function is a condition that the function requires to be true when it is called. The calling function must assure that this condition holds. The called function implementation may assume that the condition holds without further checking it. If a precondition does not hold, the called function's behavior is undefined.

If you want to make your programs more robust and to locate errors in the test phase, the functions your program calls should check to ensure that their preconditions hold. The Collection Class Library enables this checking through macro definitions. Because this checking often requires significant overhead, it is turned off by default. You need only use it while you are testing the system and verifying that preconditions are always met.

A call to a function that violates the function's preconditions has two possible results:

- If the called function checks its preconditions, the function will throw an exception.
- If the function does not check its preconditions, the behavior of the function is undefined.

Precondition-Violations

The Collection Classes include the following precondition-violation exceptions:

- IChildAlreadyExistsException—Occurs when you try to add a child to a tree using addAsChild at a position that already contains a child.
- ICursorInvalidException—Two cursor properties may lead to the ICursorInvalidException:

  1. Every time a cursor is created, you must specify the collection that it belongs to. If a function takes a cursor as an argument (such as add, setToFirst, and locate), the function can only be applied to the collection that the cursor belongs to. If the function is applied to another collection, the ICursorInvalidException results.

  2. If a function takes a cursor as an input argument (such as elementAt, removeAt, and replaceAt), the cursor must be valid. A cursor is valid if it actually refers to some element contained in the collection. You can use the isValid function to determine if a cursor is valid.

- IEmptyException—Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include firstElement and removeFirstElement.
- IFullException—Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include add and addAsFirst.
- IIdenticalCollectionException—Occurs when the function addAllFrom is called with the source collection being the same as the target collection.
- IInvalidReplacementException—Occurs when, during a replaceAt function, the replacing element has different positioning properties than the positioning properties of the element to be replaced.
- IKeyAlreadyExistsException—Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include add and addAllFrom.
- INotBoundedException—Occurs when the function maxNumberOfElements() is applied to a collection that is not bounded.
- INotContainsKeyException—Occurs when the function elementWithKey is applied to a collection that does not contain an element with the specified key.

Other Exceptions

The following lists the causes of the other exceptions defined or used by the Collection Classes:

- IOutOfMemory—Occurs when a function cannot obtain the space that it requires. This exception is not the result of a precondition violation. Functions that add an element to a collection, including add and addAsFirst, can cause this exception.
- IPositionInvalidException—Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include elementAtPosition, removeAtPosition, and setToPosition.
- IRootAlreadyExistsException—Occurs when the function addAsRoot is called for a tree that already has a root.
- ICollectionResourceException—Occurs when the Collection is constructed and the creation of the internal Resource object fails.
- ICollectionLockException—Occurs when an internal lock request fails.
- ICollectionUnlockException—Occurs when an internal unlock request fails.
- ICollectionLockTimeOutException—Occurs when a time-out value is specified during the construction of a Guard object and the lock cannot be obtained within the specified period of time.

Exceptions Caused by System Failures and Restrictions

System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur. These exceptions need to be checked for, and an exception should be thrown if they occur.

Support for Notifications

The Collection Classes include special classes that support notifications. For every concrete flat collection class (for example ISequence), there is a corresponding notification-enabled collection class (for example IVSequence).

All collection methods that modify a collection send notifications to observers. The class IVCollection defines four notification IDs for collection classes:

- addId—Sent if an element is added to the collection
- removeId—Sent if an element is removed from the collection
- replaceId—Sent if an element is replaced in the collection
- modifyId—Sent if a collection is changed in any way other than those mentioned above.

For notifications addId, removeId and replaceId, you can use INotificationEvent::eventData to access event data generated by collections. This event data is an object that includes a cursor method to access a collection cursor. The cursor points to the element referred to by the modification method. For example, if addId is the notification, the cursor points to the added element. The replaceId notification also gives you access to a copy of the element that was replaced.

Collection notifications addId, removeId and replaceId pass a pointer to the class IVCollectionEventData.

For the notifications addId and modifyId, the library sends notification after the modification occurs. For the notification removedId and replaceId the library sends notification before the collection is changed, otherwise you would not be able to use the cursor to refer to the element being removed.

Notifications are only sent if the collection is changed by the method. The following methods do not create a notification:

- removeAll for an empty collection
- add, when add does not actually add an element (for example, because the element already exists in a unique collection, or because the collection is full)
- remove if the element is not in the collection
- locateOrAdd if the element is already in the collection

Thread Safety and the Collection Classes

Like most of the IOC classes, the collection classes require thread safe operation of multithreaded access to global data. The collections may be used in a multithreaded environment, but any single instance of a collection may be referenced by only one thread at a time. Different collections may be accessed concurrently. For example, thread 1 references collection A concurrently with thread 2 referencing collection B.

The collection classes offer built-in Guard objects to simplify the explicit serialization needed to protect the collection instance. While serialization for global data is still necessary, the built-in support helps to reduce the amount of programming required. Note, however, that the locking of elements stored within a collection is the responsibility of the user and is not provided as part of collection class thread safety.

Restrictions

The current implementation does not provide any means to support users who want to program in a multiprocessing environment with the IBM Open Class collection classes having collection instances in shared memory regions executed. It only supports thread safety within a single process.

AIX Due to inherent POSIX limitations, the IBM Open Class collection classes do not support the described time-out processing in AIX environment.

Smart Pointers

In C++, variables and function arguments have their values copied when they are assigned. This copying can decrease a program's efficiency, especially when the objects are large. To improve efficiency, pointers or references are often used for common objects. For example, a pointer or reference to the object can be copied, instead of the object itself.

For efficiency, pointers to elements can be used as collection element types, rather than the elements themselves. References are not allowed as collection element types.

The Collection Classes define five pointer classes, called *smart pointers*. You can store these smart pointers, as well as standard C++ pointers, as elements in any collection:

- IElemPointer
- IAutoPointer
- IAutoElemPointer
- IMngPointer
- IMngElemPointer

Smart Element Pointers

IElemPointer, IAutoElemPointer and IMngElemPointer are special smart pointers that are designed to be kept as elements in a collection. If you store standard C++ pointers in a collection, the collection performs all element and key-type functions, except assignment, on the pointers themselves rather than on the object to which the pointer refers. This is not always what you intend. If you want the collections to perform element functions, such as equality test, on the referenced elements instead, use one of the smart element pointers (one of the classes named IxxxElemPointer). They are objects which behave as pointers to the actual element. With the element pointers, the elements themselves are not stored in the collection, although information from the elements is used by Collection Classes functions.

Automatic Storage Management

IAutoPointer, IAutoElemPointer, IMngPointer and IMngElemPointer perform storage management, which means that under certain conditions they automatically delete the object to which they refer. Automatic pointers (IAutoPointer and IAutoElemPointer) automatically delete the referenced object when the automatic pointer instance is destructed or the automatic pointer is used to point to another object. Managed pointers (IMngPointer and IMngElemPointer) keep a reference

count for each referenced object. The referenced object of a managed pointer is deleted only when the last managed pointer to the object is destructed.

IElemPointer does not provide automatic storage deallocation.

Generally Useful

Automatic storage management is particularly useful when functions return pointers or references to objects that they have created (dynamically allocated), and the last user of the object is responsible for cleaning up.

To exploit the advantage of memory management, you can use non-element smart pointers (IAutoPointer and IMngPointer), instead of standard C++ pointers, without storing them in a collection. They behave similarly to standard C++ pointers. For example, if you check the equality of two such pointers from your collection of pointers, true is only returned if the pointers point to the same address (this is the same behavior as you would expect for native C++ pointers).

Chapter 3. Implementation Variants

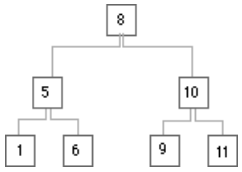
AVL Tree

AVL trees are a special form of binary tree. You can better understand AVL trees if you know how a binary tree is structured.

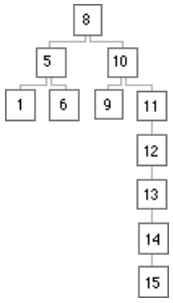
Trees are binary trees when all nodes have either zero, one, or two children. Binary trees are often used in applications where you want to store elements in a certain order. In such cases, the left child always points to an element that comes earlier in the order than the parent node, and the right child points to an element that comes later than the parent. A search through a binary tree begins at the root node. The search then continues downward until the desired element is found, by determining whether a node comes before or after the searched-for node, and then following the appropriate branch. For example, the binary tree shown in the figure below has elements added in the following sequence: 8 - 10 - 5 - 1 - 9 - 6 - 11. A search for element 9 begins at the root node (element 8). Assuming that the element value defines the ordering relation, the search would take the right node from element 8 (because 9 is greater than 8) and would arrive at element 10. The search would take the left node from element 10 (because 9 is smaller than 10) and would arrive at element 9, the desired element.

The following figures show you a binary search tree and an unbalanced binary search tree.

Balanced Binary Search Tree



Unbalanced Binary Search Tree



One drawback of a binary search tree is that the tree can easily become unbalanced. The figure shows how unbalanced the tree becomes when the elements 12 through 15 are added. The unbalanced binary tree looks almost like a list, without the performance advantage of a normal binary search tree. To obtain this performance advantage, a binary search tree should always remain balanced. The AVL Tree is a special form of binary search tree that maintains balance.

AVL trees are useful for collections containing a large number of small elements. An AVL tree implementation is even suitable for adding and deleting, because the performance overhead for the rebalancing that sometimes occurs when an element is added or deleted is still less expensive than searching through the elements of a sequence to find the position at which to add or delete an element.

If you use a set collection and do not choose an implementation variant, you are automatically using an AVL tree. If you use a set and are not aware that the set is implemented as an AVL tree, you may be surprised that a set requires an ordering

© Copyright IBM Corp. 1998

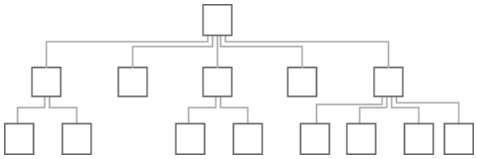
relation, when a set is an unordered collection. The reason a set requires an ordering relation is that an AVL tree requires an ordering relation to determine where to add new elements or where to find elements to be accessed or deleted. As this example shows, required element and key-type functions are determined by two factors:

- Some functions are required because of the properties of the collection.
- Some properties are required because of the implementation variant you choose.

B* Tree

A B* tree is a search tree that may have more than two references per node.The figure below shows a B* tree with up to five children per node.

A B* tree



A B* tree combines the advantages of binary search and sequential access upon the same set of keys. B* trees are based on two simple ideas:

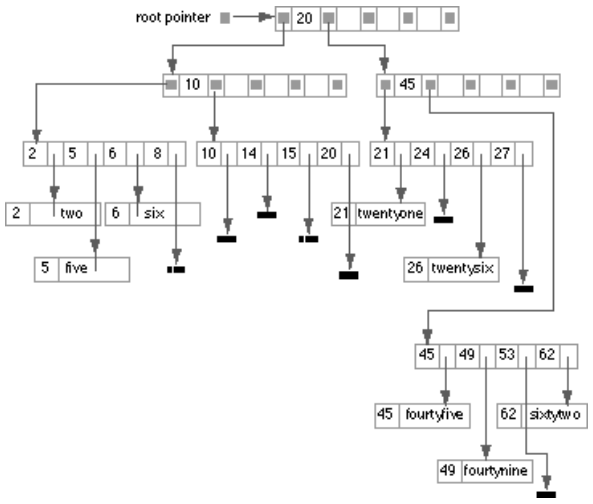
- The internal nodes are used only for storing the keys, with all real data stored at the leaves. A B* tree takes into consideration the page or block size of the operating system's virtual memory structure, and is suitable for applications where paging or memory thrashing is a constraint.
- The leaves of a B* tree are chained together in logical sequence to support sequential access.

A B* tree implementation variant is suitable when you have many large elements that are accessed by key. Because keys and their data are separated, the keys in the tree structure are used for a quick search and the pointers are used for quick access to the data.

In contrast to a B* tree, keys and data in an AVL tree are both stored in the nodes. This means that searching through elements could cause page faults if the elements are large, because the various keys may be spread across several pages along with the data they refer to.

In the figure below, the B* tree has an order of 5 (which means that each internal node has a maximum of five references). The data is stored only in the leaves. A leaf block is built to hold one element. A leaf block may be larger than one page. The B* tree implementation uses the keys in the nodes for quick access to a required page (leaf), or it uses the keys for a quick sequential access to all pages, and hence to all elements.

B* Tree Implementation Variant



Diluted Table

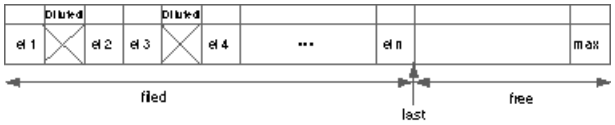
A diluted table, like a table sequence, is an array implementation of a list. However, when you delete an element from a diluted table, it is not actually deleted, but only flagged as deleted. This provides a performance advantage, in that elements following a deleted element do not need to be shifted. The additional overhead of using a dilution flag is trivial.

If you want to add a new element at a certain position, only those elements between that position and the next element flagged as deleted need to be shifted. (If no elements later in the list are flagged as deleted, then all elements beyond the insertion position must be shifted.)

Use a diluted table rather than a table if your application will be doing much adding or deleting of elements after the collection is established.

The figure below shows a diluted table implementation variant.

Diluted Table Implementation Variant



Hash Table

Hashing is another important and widely used technique for implementing collections. Conceptually, hashing involves calculating an index from the key or other parts of an element, and then using that index to look for matches in a hash table. The function that calculates the index is called a hash function.

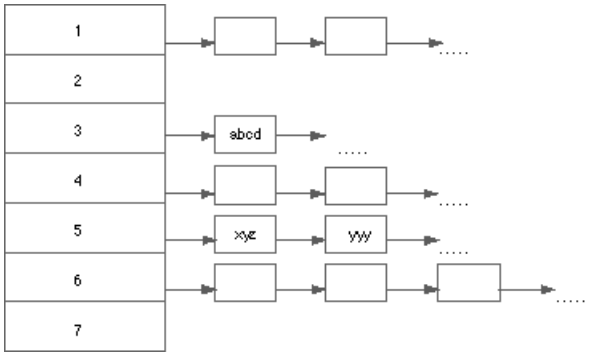
A hash-table implementation variant is suitable for nearly all applications with a balanced mix of operations. Such an implementation is quick for retrieving elements. It can also add and delete elements quickly, because, unlike an AVL tree, it does not need to be rebalanced. The efficiency of a hash-table implementation is largely dependent on the hash function implementation.

You cannot use a hash-table implementation variant when you require your elements to appear in main storage in sorted order (where elements earlier in the sorting order have lower addresses than elements later in the sorting order). On the other hand, you must use a hash table if you have a complex key (one composed, for example, of several attributes of an element), and either you cannot find a reasonable way to compare keys, or the comparison would be expensive.

For collections that do not provide access by key, but that support a hash-table implementation variant, the complete element is used as the input to the hash function.

Hashing, as implemented in the collection classes, allows elements to be stored in a potentially unlimited space, and therefore imposes no limit on the size of the collection. The figure below shows a hash-table implementation variant.

Hash-Table Implementation Variant



The hash function that calculates the index 3 from abcd is implemented as follows:

1. Each character is transformed into an integer according to its position in the alphabet.
2. The resulting integers are added together.
3. The result is divided by the hash table size. The remainder is the hash.

The principal behind a hash table is that the possibly infinite set of elements in your collection is partitioned into a finite number of hash values (1, 2, 3, ...). Your hash function is called with a key and a modulo value, and you use the key and the modulo value to arrive at an integer hash value. If for two different keys the hash function returns the same hash value (as for xyz and yyy in the previous figure), a hash collision occurs. In such cases, a hash implementation constructs a collision list where all keys returning the same hash value are linked.

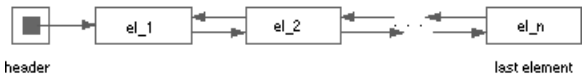
In the best case, for each different key, your hash function should return a different hash value. At the very least, it is desirable for the collision lists to remain small so that access time is fast. This means that hash values should be evenly distributed. Your hash function should randomly hash the key so that the hash value is not dependent on the key value in any trivial way. Your hash function should always return the same hash value for a given key and modulo provided to it.

List

A list uses pointers to link each element to its predecessor and successor. This implementation does not require contiguous memory for storing an array, which means that elements do not have to be shifted to make room for new elements or to close up gaps created by deleted elements.

Because storage is dynamically allocated and freed, this implementation variant is a good choice in applications that add or delete many elements, particularly where you cannot predict the amount of storage required. The figure below shows a list implementation variant.

List Implementation Variant



Table

A table is an array implementation of a sequence. The elements are stored in contiguous cells of an array. In this representation, a list can easily be traversed, and new elements can easily be added to the tail of the list. If an element needs to be inserted into the middle of the list, however, all following elements need to be shifted to make room for the new element. Similarly, if an element needs to be removed from the list, and the element is not the last element in the list, all elements following the element to be deleted must be shifted in to close up the gap.

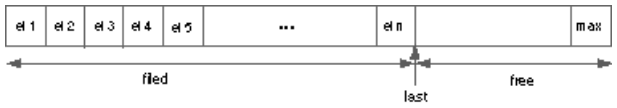
A table can access all elements quickly because all elements can be stored in a single storage block. If all of the following conditions hold true for your use of a collection, a table is a suitable implementation variant to use:

- The elements to be stored are small.
- You can predict with some accuracy how many elements your application will have to handle.
- Few or no elements will need to be added or deleted once the collection is first created.

Note that memory is statically allocated for tables, at the beginning of your program.

The figure below shows a table implementation variant.

Table Implementation Variant



Chapter 4. International Framework

International Text and Applications

The International Framework provides classes for creating international applications and manipulating international text. International applications are those that can be localized for different languages and geographic regions without accessing the source code. International text may consist of translated text strings as well as different character sets.

Locales

A locale is typically defined by a combination of language and geographical region, but other elements, such as date, time, currency, and number formats, may also help to form the locale definition.

The locale mechanism provides access for localizable resources. Localizable resources include any objects—such as transcoders, collation objects, number formatters, date formatters, and time zones—that might change to support different languages or geographic regions. For example, collation ordering rules for text sorting are associated with particular languages, while date, time, and currency formats are more closely associated with particular regions.

ILocaleKey lets you access both host-specific and portable resources associated with a particular locale. Neither ILocaleKey nor the particular locale in question owns the objects associated with it.

Collation

Collation classes enable both language-sensitive and language-insensitive string comparison. For language-sensitive comparison of Unicode text strings, the collation classes use the alphabetical ordering rules of a natural language instead of the numeric values of the character encoding system.

Transcoding

Using a set of mapping rules to convert text data between two character encoding standards is called transcoding. The Open Class transcoding classes enable conversion of Unicode character data to and from other character encoding systems, including ASCII and the code pages used in Microsoft environments. The transcoding classes also handle the conversion of special characters.

The LOCPATH environment variable identifies the location of international resources on your system. If LOCPATH is not set before using these classes on AIX or OS/2, then the 'C' locale is used. You do not need to set this variable on Windows platforms.

Locales in the Internationalization Framework

Locales provide the mechanism for organizing and accessing localizable resources—any objects that might change to support different languages or geographic regions. Such objects can include transcoders, collation objects, number formatters, data formatters, and time zones, among others.

© Copyright IBM Corp. 1998

Object Portability

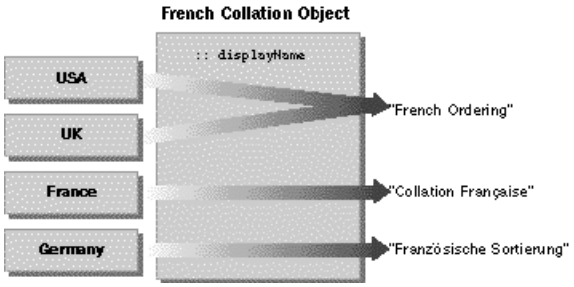
Localized objects fall into one of two categories:

- *Host* objects are specific to a particular platform implementation. You cannot typically build or modify them programmatically, nor can you stream them to other systems. They are usually tied to a specific locale, and you construct them using the key for that locale.
- *Portable* objects are not host-dependent. You can typically build them programmatically from data, edit them, and stream them to other systems. You can also construct them using a locale key if they have been associated with that locale. Portable objects can be shared between locales, but they are not required to be associated with any locale.

No locale *owns* the objects associated with it; the locale simply provides a way to access those objects.

Names of International Objects

The locale mechanism allows each locale and each object associated with that locale to have a set of displayable names. At a minimum, each object has a name for its locale in the default language and in English. Objects can also have names in other languages for display to end users, as illustrated in this figure:



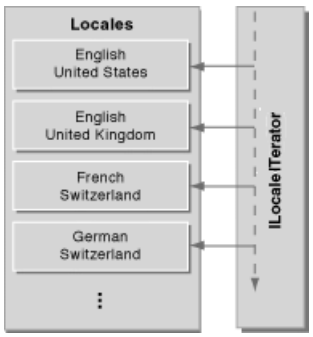
Locale Classes

This file contains information about the following subjects:

- Access to Locale Resources
- Locale Iteration

The locale classes consist of ILocaleKey**,**which gives you access to the resources associated with a particular locale, and ILocaleKeyIterator**,**which lets you iterate through the locally installed locales.

This figure illustrates how locales can be grouped by language (as for English) or by geographical location (as for Switzerland):



Access to Locale Resources

The primary class you use to work with locales is ILocaleKey. ILocaleKey acts as an identifier for a specific locale and is used as a parameter for functions that create a particular type of localizable object. For example, the ICollation::createCollation function takes an ILocaleKey object that indicates what language the collation should reflect. On the other hand, the ILocaleKey::displayName function lets you display the locale name in any language for which a name is provided. Each type of international object should provide a displayName function, such as ICollation::displayName.

When you need a localized object, create an ILocaleKey object to define the locale with which to associate the object. A given object can be associated with more than one locale. Thus, a collation object for English could be associated with many—or even all—English-speaking locales (United States, Canada, United Kingdom, Jamaica, and so on). The set of objects associated with a locale can be open-ended; you are free to associate new objects with the locale.

Use ILocaleKey to identify the locale for which you want to access localized resources. You can use ILocaleKey to specify a particular locale (such as U.S. English) or to specify the default locale on the current host. ILocaleKey gives you access to both host-specific and portable objects associated with a given locale.

ILocaleKey supports two ways to describe a specific locale:

- Using the host locale ID (an `unsigned long` value) for the current platform. For example, on Win32 platforms the locale identifier for United States English is 0x0409.
- Using the portable POSIX identifier. This has three parts:
  - A two-character language identifier, for example, 'EN' for English or 'ZH' for Chinese.
  - A two-character region identifier, for example, 'US' for the United States or 'TW' for Taiwan.
  - A single-character extension used in a few cases where the standard 'language_region' POSIX identifier maps to multiple, distinctive locales. For example, the 'NO_NO' identifier maps to both the BokmÔl Norwegian and Nynorsk Norwegian locales. An extension lets you provide a unique identifier for each.

When you use a locale key to obtain an object for a given locale, the system does the following:

1. If there is a corresponding host object for that locale, the system returns it.

2. If there is no corresponding host object for that locale, the system looks for an object that corresponds to the same language.
3. If there are no corresponding host objects, the system looks for a corresponding portable object.
4. If there is no corresponding portable object, the system throws an exception.

This figure shows the interface for ILocaleKey:



- languageID and setLanguageID let you access the language identifier portion of the locale ID. Two special language identifiers, 'C' and 'POSIX,' indicate the POSIX C locale. If you specify one of these language identifiers, the region and extension are set to null. If you set the language identifier to a null string, the locale key is set to the default host locale.
- regionID and setRegionID let you access the region identifier portion of the locale ID.
- extension and setExtension let you access the extension used to uniquely identify locales whose POSIX IDs map to more than one locale.
- posixID and setPosixID let you access the entire POSIX identifier string. This string has the format 'fLanguageID_fRegionIDfExtension,' for example, 'ES_ESM.'
- hostID and setHostID let you access the host identifier. *(hostID and setHostID are supported only on Windows platforms.*)
- isCLocale indicates whether the locale key references the C (POSIX) locale.
- defaultLocale is a static function that returns a locale key for the current default locale.
- displayName returns a localized, displayable name for the locale. You specify the locale in which to display the name and a lookup strategy for finding the name:
  - kExactLocale—Return the display name only for the exact locale specified.
  - kExactLanguage—If there is no name for the exact locale specified, return the name from another locale with the same language identifier.
  - kAnyLanguage—If there is no name in the specified locale or language, return the name in the default locale and language. This is the default option.

Locale Iteration

Use ILocaleKeyIterator to iterate through the locales available on the current hosts. Because ILocaleKeyIterator iterates through currently installed locales (rather than all locales supported by the locale mechanism), it may not reflect the full set of supported locales. You do not need to initialize an ILocaleKeyIterator object—when constructed, it automatically references the first locale key in the list.

This figure shows the interface for ILocaleKeyIterator:



- operator* returns the locale key referenced by the iterator.
- operator-> returns a `const` pointer to the locale key referenced by the iterator.
- operator++ and operator— advance the iterator to the next or previous locale key.
- reset sets the iterator to reference the first locale key.
- operator bool and end return `true` if the iterator references the last locale key.

Collation Classes

This file contains information about the following subjects:

- Overview of Collation Classes
- Collation Subclasses
- Collation Iteration
- Ordering Strength

Overview of Collation Classes

In most cases, the ordering of Unicode values does not produce correct ordering results. For example, in the ASCII-based character sets, *Z* is ordered before *a*, and *z* is ordered before ±. Open Class collation classes, however, support collation objects that compare strings based not on the Unicode values of each character, but on the rules of a natural language. This is what enables language-sensitive string comparison.

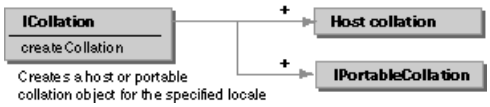
Each International Framework collation objects is based on a set of rules that define the results for alphabetizing and comparing text in a particular natural language. These rules define not only a ranking (such as a < b < c) but three levels of priority within the ranking.

For many European languages, the difference between two base letters (*a* and *b*) is a primary difference, the difference between an unaccented and an accented base letter (*õ* and *a*) is secondary, and the difference between an uppercase and lowercase letter (*A* and *a*) is tertiary. These distinctions allow you to set the level of comparison for more sophisticated sorting and searching.

The ICollation interface is based on the protocols in the ANSI C++ standard library `collate` class, which provides string comparison and hashing functions. The ICollation comparison functions take two strings or substrings and return a value that indicates whether the source string is greater than (later in the alphabet), less than (earlier in the alphabet), or equal to the target string. You can specify the ordering strength of the comparison to control how differences such as case and accents are handled.

You can compare styled text in an IText object, but styling information is ignored.

This figure shows the collation class architecture:



Collation Subclasses

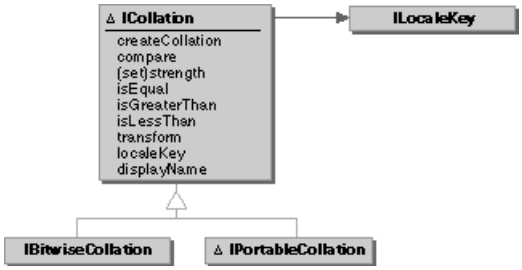
The collation classes include the abstract base class ICollation, which defines the protocol for language-sensitive string comparison and several concrete subclasses, and ICollationIterator, which lets you iterate through the list of available localized collation objects.

Class	Description
IBitwiseCollation	Provides bitwise, language-insensitive string comparison.
ICollation	Provides access to either host-specific or portable collation for a given language as available. Primary class for language-sensitive string comparison.
ICollationIterator	Lets you iterate through the available collation objects.
IPortableCollation	Provides portable (non-host-specific) language-sensitive collation.

ICollation provides the protocols you use to create both language-sensitive and language-insensitive collation objects. The ICollation interface is a superset of the interface of the ANSI C++ Standard `collate` class. Based on the locale you specify, the ICollation::createCollation function can return:

- A host-specific collation object for the specified language or locale
- A portable collation object for the specified language or locale, derived from IPortableCollation
- To specifically create an IPortableCollation object, call IPortableCollation::createCollation directly with a portable locale key
- An IBitwiseCollation object that performs language-insensitive collation
- To request that createCollation return an IBitwiseCollation object, specify the POSIX locale ('POSIX' or 'C').

This figure shows the ICollation interface:



- createCollation is a static function that returns the collation object for a specified locale. If a host-specific object is available, it is returned. Otherwise a portable object is returned. If you don't specify a locale, the function returns the collation object for the default locale. createCollation also lets you specify a comparison level. The default is ICollation::kTertiaryDifference.

- compare returns the result of comparing two strings. The result is returned as an enum value: kSourceEqual, kSourceLess, or kSourceGreater.
- strength and setStrength provide access to the collation object's current ordering strength (primary, secondary, or tertiary).
- isEqual, isGreaterThan, and isLessThan are convenience functions that return a Boolean value indicating the comparison result of two strings.
- transform converts an IText into another IText that is compared lexicographically with the original text. Comparing two transformed IText objects returns the same results as comparing the same strings before transformation.
- localeKey returns an ILocaleKey indicating the locale the collation object is associated with.
- displayName returns a displayable name for the object for a specified locale.

Collation Iteration

Use ICollationIterator to iterate through the list of international collation objects currently available on the system. You can set the iterator to enumerate only host collation objects, only portable collation objects, or both (both is the default).

This figure shows the interface for ICollationIterator:



- operator++ increments the iterator to the next collation object.
- operator bool and end indicate that the iterator is at the end of the list.
- reset sets the iterator to reference the first item in the list.
- create returns the collation object that the iterator currently references. You can specify the ordering strength.
- localePOSIXID returns a text string that contains the POSIX ID of the locale associated with the currently referenced collation object.
- displayName returns a displayable name for the object for a specified locale.

Ordering Strength

The correct collation for each language or script is determined by a set of rules that define a ranking, from least to greatest, for each character. To allow more comparison options, each character is assigned an ordering priority within the ranking: primary, secondary, or tertiary. For example, in an English collation:

- Base letters represent a primary difference ('a' and 'b')
- Diacritical marks on the same base letter represent a secondary difference ('a' and 'Ô')
- Uppercase and lowercase versions of the same base letter represent a tertiary difference ('a' and 'A')

In English, then, you can implement case-insensitive comparison by setting the ordering strength to kSecondaryDifference. Primary and secondary differences are considered but any tertiary (case) differences are ignored-thus, 'pat,' 'Pat,' and 'PAT' would be considered equivalent strings.

When you create a collation object, you specify an ordering strength that determines whether all differences, both primary and secondary differences, or only primary differences are considered. The types of differences that are considered primary, secondary, and tertiary may vary based on the language you are working with.

This table shows the results for English strings compared with different ordering strengths:

Source	Target	Ordering strength	Comparison result
abc	abc	kPrimaryDifference	kSourceEqual
õbc	abc	kSecondaryDifference	kSourceEqual
Abc	abc	kTertiaryDifference	kSourceEqual
abc	def	kPrimaryDifference	kSourceLess
abc	õbc	kSecondaryDifference	kSourceLess
abc	Abc	kTertiaryDifference	kSourceLess
def	abc	kPrimaryDifference	kSourceGreater
õbc	abc	kSecondaryDifference	kSourceGreater
Abc	abc	kTertiaryDifference	kSourceGreater

When you are using the collation object for the POSIX locale (a portable collation) specifying an ordering strength has no effect.

Transcoding Classes

This file contains the following subjects:

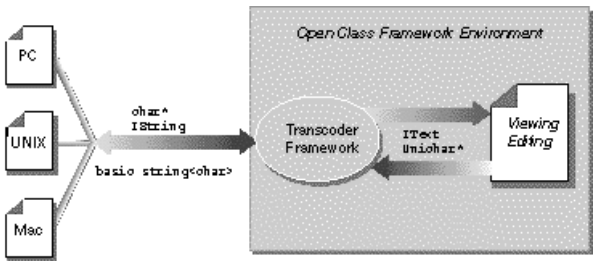
- Overview of Transcoding Classes
- Transcoders
- Line Breaking
- Character Set Iteration

Overview of Transcoding Classes

Transcoding is the process of converting text data between two coded character sets using a mapping rules.

The goal of the Open Class libraries is for all text to be encoded in Unicode and manipulated according to the Unicode character encoding standard. Any text you import from or export to a system that uses a different character encoding scheme must be transcoded so that the text can be manipulated directly on the target system.

The Open Class transcoding classes support conversion of character data to and from Unicode and a wide variety of other encoding sets and encoding schemes, including ASCII and other ISO standards, to enable you to import and export text data between Open Class applications and other environments. The Open Class transcoders use the `UniChar` datatype to represent Unicode characters in IText objects and the `char` datatype to represent non-Unicode characters in IString objects.



The transcoding classes also provide mechanisms for handling characters that don't have obvious mappings between Unicode and another character set. These mechanisms handle both line-breaking characters, which differ between platforms, and exception characters. Exception characters are characters that can often be transcoded but do not have a one-to-one mapping. These may include ligature characters, foreign characters, or composed characters.

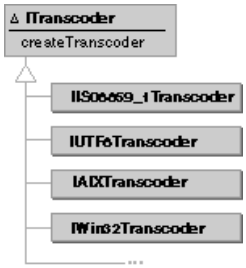
The transcoding classes are:

Class	Description
ITranscoder	Primary class defining protocols for transcoding character data between Unicode and any other character encoding standard
ILineBreakConverter	Simple class used to postprocess line-breaking conventions after character data is transcoded into Unicode, or preprocess line-breaking characters before Unicode data is transcoded into `char`-based data
ICharacterSetIterator	Lets you iterate through the character sets for which transcoders are available

Transcoders

ITranscoder provides the abstract protocol for all transcoders supported by the Open Class system. You create a transcoder by specifying a character set to ITranscoder::createTranscoder, which returns an instance of the ITranscoder subclass supporting that character set.

This figure shows some of the available transcoders:



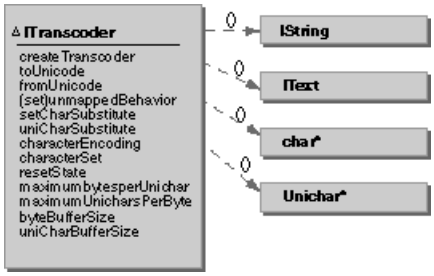
ITranscoder is currently the only public transcoder class. You access all concrete subclasses through the ITranscoder interfaces.

ITranscoder provides both a simple, high-level interface, which converts between IText and IString instances, and a low-level interface, based on the ANSI C++

Standard codecvt interface, which takes pointers to char and UniChar arrays. The high-level functions take two parameters, a char-based IString object and a UniChar-based IText object, and convert either to or from Unicode data.

The low-level pointer-based functions let you manipulate char and UniChar strings directly. Some of the low-level functions are identical to the interfaces provided by the ANSI C++ standard library codecvt class. They allow you to specify exact ranges of text to transcode and to provide error-recovery mechanisms.

This figure shows the ITranscoder interface:



- createTranscoder returns a transcoder for the character encoding set you specify. See the table 'Open Class transcoder names' for a list of supported transcoder names. If you don't specify a name, the function returns a transcoder for the current host character set.
- toUnicode and fromUnicode provide conversion between char and UniChar data. Overloads of these functions take IString or char* and IText or UniChar*.
- result returns an enum value that indicates whether the conversion was fully converted, partially converted, or stopped due to an error. This value is also returned by toUnicode and fromUnicode.
- unmappedBehavior and setUnmappedBehavior let you determine how the transcoder handles exception characters.
- setCharSubstitute lets you specify a character to be used as a substitute for characters that do not have a mapping from Unicode to the specified character set. The default character is UASCII::kSubstitute (0x1A).
- uniCharSubstitute returns the character used as a substitute for characters that do not have a mapping from the source character set into Unicode.
- characterEncoding returns an IText containing the name of the character encoding supported by the transcoder.
- characterSet returns an IText containing the name of the default encoding for a specified locale.
- resetState resets the state of the transcoder to ASCII.
- Storage query functions let you get information about storage requirements so you can manage storage allocation for transcoding operations efficiently.

Line-Breaking Conversion

ILineBreakConverter is a simple class that you use to ensure that line breaking characters are transcoded correctly between Unicode and the target character set. You can use ILineBreakConverter to postprocess strings just converted into Unicode, or preprocess strings before converting them into char data.

This figure shows the ILineBreakConverter interface:



- convertInPlace and convert process the line breaks in an IText object according to a specified line-breaking convention.
- hostConvention returns the line-breaking convention for the current host.

ILineBreakConverter uses the enum ELineBreakConvention to describe different line-breaking conventions. Currently the following conventions are defined:

Line-Breaking Conventions

kUnicode	Unicode convention	UGeneralPunctuation::kParagraphSeparator (U+2029)
kCRLF	Windows, DOS, OS/2 convention	CR LF sequence
kLF	UNIX convention	LF
kCR	Macintosh System 7 convention	CR
kCRLF_VT	Microsoft Word/Rich Edit convention	CR LF or VT
kHost	Indicates the current host's convention	

ILineBreakConverter uses the following rules to convert between various host line-breaking conventions and Unicode:

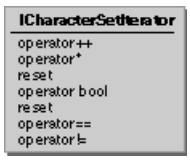
Line-Breaking Conversion Rules

Host	Line-breaking convention	Unicode text with host convention	Unicode convention
Win32, OS/2, DOS	CR LF sequence	0x000D 0x000A	U+2029
AIX	LF	0x000A	U+2029
Macintosh	CR	0x000D	U+2029
Word/RichEdit	CR LF sequence	0x000D 0x000A	U+2029
Word/RichEdit	VT	0x000B	U+2028

Character Set Iteration

Use ICharacterSetIterator to iterate through the list of character encoding sets for which transcoders are available on the current system. ICharacterSetIterator returns IText objects that contain the names of supported character sets.

This figure shows the interface for ICharacterSetIterator:



- operator++ increments the iterator to reference the next available transcoder.
- operator* returns an IText with the name of the character set supported by the currently referenced transcoder
- operator bool indicates when the iterator has reached the end of the list.
- reset sets the iterator back to the first available character set

- operator== and operator!= are equality and inequality operators for two character set iterators

Special Characters

The ITranscoder transcoding functions provide special handling for both line-break and exception characters.

The class ILineBreakConverter provides for conversion between the Unicode paragraph-separator character (U+2029 or UGeneralPunctuation::kParagraphSeparator) and the appropriate line-break character for a given character set or host. You can use this class to postprocess transcoded strings after conversion into Unicode or to preprocess strings before conversion into `char`-based formats.

ITranscoder also lets you control how exception characters are handled. Exception characters are characters that do not have a single-character equivalent or that do not exist in the target character set. For example, Greek characters may be used in some environments where they are not part of the native character set, and ligature characters, which by definition combine two characters, are often mapped to a sequence of their individual components. The following table shows some typical cases:

Unicode Name	Unicode sequence	Display	May be mapped to	Control code
LATIN SMALL LIGATURE FI	FB01	[fi]	0066 [f] + 0069 [i]	\xde
GREEK CAPITAL LETTER DELTA	0394	D	² or other	\xc6
GREEK SMALL LETTER PI	03C0	p	* or ⅄ or other	\xb9

To specify how you want exception characters to be handled, call the ITranscoder::setUnmappedBehavior function. You can specify a substitution character, in either Unicode or the target character set, or you can specify that the transcoder either skip exception characters or stop the transcoding operation when it reaches one.

By default, transcoders substitute UGeneralFunction::kReplacementCharacter (U+FFFD) for Unicode characters with no mapping, and the ASCII substitution character (UASCII::kSubstitute, or 0x1A) for `char` characters with no mapping.

Exception Characters

The transcoders let you specify how you want exception characters to be handled. Exception characters are characters for which there are no one-to-one mappings between Unicode and the target character set. Use the EUnmappedBehavior enum to specify one of the following:

kUseSub	Substitutes an equivalent representation in the target character set for characters with no exact mapping
kStop	Stops transcoding when an exception character is detected

kOmit	Skips any exception characters detected

If you don't specify behavior for exception character handling, the transcoder uses EUnmappedBehavior::kUseSub as the default. The substitute characters that are used are:

- UGeneralPunctuation::kReplacementCharacter (U+FFFD) for characters that cannot be transcoded into Unicode
- UASCII::kSubstitute (0x1A) for characters that cannot be transcoded out of Unicode, that is, into the target `char`-based character set

You can set the `char` substitute character to another character with the ITranscoder function setCharSubstitute. Whether to display these characters as glyphs or as text strings is left to the host operating system.

Mapping Proximity

When you create a transcoder for a specific character set, you can specify how close the mapping proximity must be between Unicode and the target character set. Use the EMappingProximity enum to specify one of the following:

kExactMapping	Create a transcoder with an exact mapping to the specified character set
kSupersetMapping	Create a transcoder with a character set that is a superset of the specified character set
kCloseMapping	Create a transcoder with a character set as close to the specified character set as possible

If you don't specify a mapping proximity, the transcoder uses EMappingProximity::kSupersetMapping as the default.

Chapter 5. Date and Time Classes

Dates and Calendars

The IDate class uses Gregorian calendar dates. The Gregorian calendar is in general use and consists of the 12 months, January to December.

IDate also supports the Julian date format, which contains the year in positions 1 and 2, and the day of the year in postions 3 through 5. If the day of the year is less than three digits, zeros are added on the left to increase the size to three digits. For example, February 14, 1965 is 65045 as a Julian date. (February 14 is the 45th day of the year.)

The IDate class returns the names of the days and months in the language defined by the current locale. For information on defining the locale, see the standard C library function setlocale().

Date and Time Classes

The IDate and ITime classes are independent classes that provide you with data types to store and manipulate date and time information. Because the IDate and ITime classes are independent, when an ITime object's time passes 23:59:59 (24-hour format) or 11:59:59 (12-hour format), it has no effect on the value of any IDate object.

The ITimeStamp class provides you with a data type to store and manipulate timestamp information, where a timestamp represents a specific point in time; for example, combined date and time.

With these classes, you can create date, time, and timestamp objects, and use member functions to do the following:

- Write date, time, or timestamp objects to an output stream
- Access detailed information about dates, times, or timestamps
- Compare dates, times, or timestamps
- Test the characteristics of date or time objects
- Add or subtract days from a date object
- Add or subtract hours, minutes, or seconds from a time or timestamp object
- Convert between date formats or between time formats.

Time

The ITime class refers to time in the 24-hour format by specifying time units (hours, minutes, seconds) past midnight. If you want to display ITime objects in the 12-hour format, you must convert them to IStrings using the asString function with a char* argument of '%r'. (This argument is a format string. All format specifiers of the strftime() function of the standard C library are supported by the IString conversion function.)

Note: Objects of the ITime class are precise only up to the nearest second, and cannot be used for more precise timings.

© Copyright IBM Corp. 1998

Time Stamps

An ITimeStamp object can be created from an IDate object, an IDate and ITime object, or a value that represents the number of seconds from the reference date 01/01/2000 00:00:00. If the time stamp is referring to a point in time before the reference date, a negative value must be used.

Chapter 6. National Language Support and Double-Byte Character Sets

Overview of National Language Support

VisualAge for C++ provides national language support using the XPG/4 programming model, and using the locale-sensitive functions of the C runtime library.

When you enable NLS, member functions of the IString, IDate, ITime, and ITimeStamp classes become locale sensitive, in both SBCS and DBCS environment. The classes provide the following capabilities:

IString
: Character string handling in SBCS and DBCS environments

IDate Date formatting and manipulation functions

ITime Time formatting and manipulation functions

ITimeStamp
: Date and time formatting and manipulation functions

While the interfaces of these classes do not change when you enable NLS, the underlying semantics change to reflect locale requirements. For example, the compare family of IString functions no longer perform bitwise comparisons, but instead perform comparisons based on the string collation sequence defined by the current locale.

Double-Byte Character Set Support

Objects of the IString class and the I0String class can contain a mixture of single-byte characters and double-byte characters. All member functions allow for the mixture. The searching functions will not match a single-byte character with the second or subsequent byte of a double-byte character. Functions that return substrings will never separate the bytes of a double-byte character.

Although the double-byte characters are supported, you must be careful not to alter the contents of a string in a way that would corrupt the data. For example, the statement:

```
IString[n]='x';
```

would be an error if the nth byte of the IString was part of a double-byte character.

DBCS and National Language Support

The Data Type and Exception Class Library provides double-byte character set (DBCS) support and national language support (NLS). You can use one source file for your application code and provide DBCS and NLS support by using separate resource files for the languages you support. The benefits of this organization include the following:

- The application is easy to maintain, because a single version of the application is used. This reduces the cost of maintaining your code.

© Copyright IBM Corp. 1998

- The application is easy to upgrade because only the source code is upgraded and then linked to the separate language files for different languages. This reduces the time and cost of upgrading your code because different language versions can be generated at the same time.

Because message strings are defined in either resource files or message catalogs, they can be translated easily to your local language without changes to the source code.

You should note the following when creating a DBCS-enabled application:

- String manipulation is DBCS-enabled. The string classes support mixed strings that contain both SBCS and DBCS characters. Use the string testing functions to determine if a character is single byte or double byte.
- The IDBCSBuffer class ensures that the search functions do not match the second or any subsequent bytes of a DBCS character and that the bytes of a DBCS character will not be split.

Chapter 7. Text Framework

Text Overview

The classes that are part of the Open Class Text Framework enable the creation, storage, querying ,and modification of text strings.

The primary classes provided by the Text Framework are:

- IText, a variable-length styled string class you can use for storing styled or unstyled international text. The IText class is optimized for text of any length, from short strings to large text data segments. The length of an IText string is limited chiefly by hardware and operating system constraints (the theoretical limit is four billion characters). For ease of use and consistency, IText stores style data along with character data so that you can use styled characters throughout the Open Class environment.
- ITextBoundary, which implements methods for locating boundaries of characters, words, lines, and sentences.
- IString, which can be useful when you do not require the style and international text capabilities provided by IText.
- Style classes, which can be applied to individual characters, ranges of characters, or paragraphs.
- Iterator classes, which provide access to the character data in IText objects.

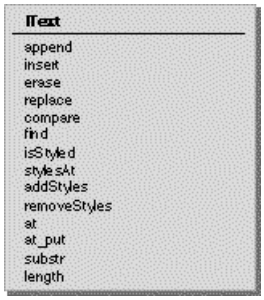
Text Creation and Manipulation

The primary interfaces you need to create and manipulate Unicode text strings, whether styled or unstyled, are provided by IText. It gives you access to the storage mechanism and provides a complete set of protocols for accessing and manipulating both the character and the style data. IText is designed so that you can use it to store unstyled text without the reduction in performance or increase in memory requirements that may be associated with the styling mechanism.

You can create an IText object from another IText, from an IString, or from an array of `char` or `UniChar`. When you convert between IText and a `char`-based format, the data is transcoded. The conversion functions assume the `char`-based format is ISO 8859-1.

The IText interface parallels the API provided by the ANSI C++ class `basic_string`. It includes functions for converting between string formats, for manipulating styles, and for working efficiently with the IText storage mechanism.

This figure is a partial listing of the IText interface:



© Copyright IBM Corp. 1998

IText also supports the `basic_string` interfaces for simple searching and comparison. These functions perform simple bitwise comparisons, and ignore styling information. For language-sensitive comparison of IText strings, use the International Framework's collation classes.

This figures shows the IText interfaces for searching and comparison:



Text Boundaries

This file contains the following subjects:

- Identifying Text Boundaries
- Character Boundaries
- Word Boundaries
- Line Boundaries
- Sentence Boundaries

Identifying Text Boundaries

The ITextBoundary class implements methods for finding the location of boundaries in text. ITextBoundary is an abstract base class. Instances of ITextBoundary maintain a current position and scan over text returning the index of characters where boundaries occur.

Character Boundaries

Character boundary analysis allows users to interact with characters as they expect to, for example, when moving the cursor through a text string. Character boundary analysis provides correct navigation through strings regardless of how a character is represented. For example, an accented character might be stored as a base character and a diacritical mark, or a single combined character. What users consider to be a character can differ between languages.

Word Boundaries

Word boundary analysis is used by search and replace functions, as well as within text editing applications that allow the user to select words with a double click. Word selection provides correct interpretation of punctuation marks within and following words. Characters that are not part of a word, such as symbols or punctuation marks, have word breaks on both sides.

Line Boundaries

Line boundary analysis determines where a text string can be broken when line-wrapping. The mechanism correctly handles punctuation and hyphenated words.

Sentence Boundaries

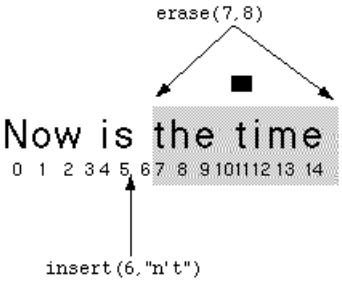
Sentence boundary analysis allows selection with correct interpretation of periods within numbers and abbreviations, and trailing punctuation marks such as quotation marks and parentheses.

Text Storage

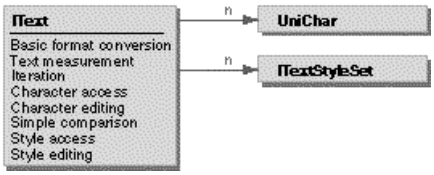
IText is the basic mechanism for storing and manipulating Unicode text strings throughout the Open Class libraries and frameworks. IText encapsulates Unicode characters and any associated styling information, and fully supports mixed style runs. IText keeps the styles with the characters, so you can pass text strings between objects and applications without loss of styling information.

IText is the primary string format supported by the Open Class International Framework.It is suitable for strings of any length, from a few characters to document-length strings. IText was also designed so that you can use it for unstyled strings without incurring the overhead in object size or performance associated with the styling mechanism.

Many IText functions take a range that specifies the subset of characters to operate on. The range is defined by an offset and a character count, where the offset of the first character in the object is 0. To specify an insertion position, specify the offset of the character immediately following the position where the new text will be inserted.



This figure shows the IText interface and related classes:



Storage Mechanism

IText manages its own storage. The characters and styles are stored in separate objects that the IText mechanism creates, deletes, and shares transparently. IText itself is very small and has very fast copy performance. This allows you to:

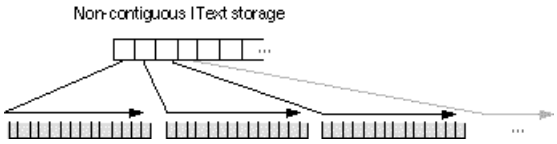
- Include IText objects directly as data members in other objects. Your classes should have pointers or references to IText objects only if you want several objects to all see a change to a particular IText object.
- Pass IText parameters to functions as values rather than references, unless the IText object is owned by another object.

- Return IText objects from functions as return values rather than with fill-in parameters, unless the IText object is owned by another object.
- Use IText objects as local variables—that is, do not use the `new` and `delete` operators.

The underlying storage object can be shared by multiple IText instances. It is reference counted and uses copy-on-write semantics. For example, when an IText object is copied, the actual storage is not duplicated—the reference count is simply increased. Note that even while they share storage, two IText objects behave as two distinct objects. They stop sharing storage when one is modified. This is guaranteed to be true even in multithreaded situations.

The framework manages this mechanism for you. However, you should be aware of it when using classes such as IFastTextIterator, which do not consider the underlying storage mechanism.

The storage mechanism handles both short and large strings efficiently. The storage allocation strategy changes dynamically as appropriate for the size of the string. For small strings, the characters are stored in a single, contiguous, heap-allocated array, resized only when necessary. Longer strings are broken up into non-contiguous storage blocks, or chunks, as illustrated in this figure:



The IText function storage_chunk provides access to these chunks of text. You specify a character offset, and the function returns a pointer to the chunk of storage containing that offset. If the string is stored in a single contiguous block, the function returns a pointer to that block.

Strings and Buffers

This file contains the following subjects:

- Overview of Strings and Buffers
- String Buffers
- String Classes
- String Comparison
- String Indexes

Overview of Strings and Buffers

You can store and manage strings using the string and buffer classes. There are two type of string classes, two types of buffer classes, and two support classes. The two string classes, IString and I0String, are the main classes. The buffer and support classes are used to implement the string classes.

The IString class provides a wide range of string handling capabilities. Many of the IString operators and functions are overloaded to support both IStrings and arrays of characters as return types and arguments. For example, the comparison

operators (==, >, <, >=, <=, !=) all support either two IString operands or one IString and one array of characters operand. The array of characters operand can be on either side of the comparison operator.

If you are using the string classes, DBCS support is nearly automatic and transparent. If the underlying operating system supports it, you can use enableInternationalization to turn DBCS support on.

The support classes, IStringEnum and IStringTest, provide data types and testing functions that are used in the string and buffer classes.

String Buffers

When you create an object of a string class, the actual characters that make up the string are not stored in the string object. Instead, the characters are stored in an object of a buffer class.

The use of a buffer object is transparent to you when using the string classes. A correctly sized buffer is automatically created when you create a string object. The buffer is destroyed when a string object is destroyed. When you manipulate or edit a string, you are actually manipulating and editing the buffer object that contains the characters of the string.

String Classes

The string classes define a data type for strings and provide member functions that let you perform a variety of data manipulation and management activities. They provide capabilities far beyond those available with standard C strings and the string.h library functions.

The string classes have the following capabilities:

- String buffers are handled automatically.
- Strings can contain both SBCS and DBCS characters.
- Strings can be indexed by character or by word.
- Strings can contain null characters. (There are no restrictions on the contents of a string object.)

Member functions of the string classes allow you to:

- Use strings in input and output
- Access information about strings
- Compare strings
- Test the characteristics of strings
- Search for characters or words within a string
- Manipulate and edit strings
- Convert strings to and from numeric types
- Format strings by adding or removing white space

There are two string classes: IString and I0String. They are identical except for the method each uses to index its characters. The characters of an IString object are indexed beginning at 1. I0String characters are indexed beginning at 0. The string class you should use depends on which indexing scheme you prefer or find easier to implement.

Objects of IString and objects of I0String can be freely intermixed in a program. Objects of one class can be assigned objects of the other. Arguments that require an object of one will accept objects of the other. You will only notice a difference between an IString and an I0String when you are using functions that use or return a character index value.

Each IString function has a corresponding I0String function with the same name. The I0String version of each function accepts the same arguments and has the same return type as the IString version except that all parameters of type IString become I0String.

String Comparison

The IStringTest class lets you define the matching function used in the searching and testing functions of the string and buffer classes. When a search string is passed to a searching or testing function, the search string and the string object are compared on a character-by-character basis. The characters are considered to match if they are identical. The IStringTest class allows you to define when characters are considered to match.

For example, you can implement a string test that locates a given occurrence of a particular character in a string:

```
//Using the IStringTest class

#include <istring.hpp>
#include <iostream.h>

class Nth : public IStringTest {
   char key;               // Specifies the character to look for
   unsigned count;         // Specifies which occurrence to find
   public:
      //
      // Construct an Nth object as follows:
      // 1. Create an IStringTest instance whose function type is user,
      //    with a null character to start;
      // 2. Initialize the count to n
      // 3. Initialize the key to c
      //
      Nth(char c, unsigned n)
      : IStringTest(user,0), count(n), key(c) {}

      //
      // test function: accepts an int (the character to look for)
      // checks if the character matches the key
      // if so, decrements count
      // eventually, count will equal zero if enough matches are found,
      // so 'return !count' will return true (-1)
      // otherwise, 'return !count' will return a value other than -1

      virtual Boolean test (int c) const
         {
         if (c == key)                // if it matches,
         ((Nth*)this)->count—;       // decrement count
         return !count;               // return complement of count
                                      // will be true (-1) if count==0
         }
};

void main() {
IString text='this is a test string';
cout << 'The fourth appearance of the letter t in the string:\n'
     << tet << '\n' << 'is at position '
     << text.indexOf(Nth('t',4)) << endl;
}
```

This program produces the following output:

```
The fourth appearance of the letter t in the string:
this is a test string
is at position 17
```

A derived template class, IStringTestMemberFn, is provided to support the use of the IStringTest class with any function that accepts its objects as an argument.

A constructor for IStringTest accepts a pointer to a C function. The C function must accept an integer as an argument and return a boolean. Such functions can be used anywhere an IStringTest can be used. Note that this is the type of the standard C library functions that check the type of C characters, for example, isalpha() and isupper().

String Indexes

Objects of the string classes are arrays of characters. There are two types of indexes used with the arrays. The first is a character index. Each character is numbered from left to right starting at the number 1 in the IString class and the number 0 in the I0String class. Therefore, in the IString 'The dog is brown', the letter 'i' has an index value of 9. In the I0String 'The dog is brown', the letter 'i' has an index value of 8.

The second type of index is the word index. In the word index, each white-space-delimited word is numbered from left to right starting at the number 1. The word index is the same for IString objects and I0String objects. Therefore in the IString 'The dog is brown', the word 'is' has an index value of 3. In the I0String 'The dog is brown', the word 'is' also has an index value of 3.

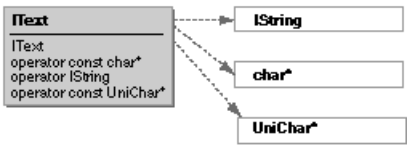
The only difference between objects of the IString class and objects of the I0String class is the starting value for the character index.

String Formats

IText provides full interoperability with the `char`-based string formats `char*` and IString, allowing you to:

- Construct IText objects from null-terminated `char` arrays and from IString objects (IText also provides constructors that take both null-terminated-terminated with UUnicodeSpecial::kNull-and non-terminated `UniChar` arrays)
- Return the characters in an IText as a `char` array or an IString object
- Use IText where `char*` or IString is called for
- Use `char*` or IString where IText is called for

This figure shows the IText interfaces supporting compatibility with other string formats:



IText uses the transcoding facility provided by the International Framework to convert data between `char` and Unicode data. IText assumes the `char`-based format is the default character encoding set for the host. This varies based on the platform and any locale settings for the system.

String Size and Capacity

IText follows the ANSI convention of using `size_t` (generally equivalent to `unsigned long`) to represent text lengths and offsets. IText defines two typedefs equivalent to `size_t`:

- length_type represents a character count. length_type values always represent the number of characters, not the number of bytes. IText also uses a special length_type value, `npos`, to represent the number of characters between a specified starting offset and the end of the string. For example, a starting offset of 0 and a length of `npos` refers to the whole string.
- offset_type represents the position of a character, zero-based from the first character in the string-that is, the position of the first character is at offset_type value 0, and so on.

This figure shows the IText functions you use to access the string's size and the storage capacity of the object:



- length and size are equivalent functions that return a length_type value representing the number of characters currently in the string.
- max_size returns the maximum number of characters that can be stored in an IText object, currently equivalent to `npos - 1`.
- resize lets you reset the size of the string to a specified number of characters, either deleting characters or padding the end of the string as necessary. The default padding character is UUnicodeSpecial::kNull.
- capacity returns the number of characters for which the IText object has currently allocated storage.
- reserve can be used to indicate that a specified number of characters are about to be added to the string. The function will preallocate storage if needed and if memory conditions permit it.
- empty indicates whether the text object contains any character data.
- c_str and data return `UniChar` arrays containing the same number of characters as the IText object. c_str always returns a null-terminated array-data may not.

IText versus IString

Earlier releases of IBM Open Class relied more on IString than on IText, which is designed to replace it. Nevertheless, IString remains useful for certain functions, such as numeric formatting and parsing, case conversion, language-sensitive or case-insensitive searching, iterative find and replace, and space padding. While IText is usually a better choice, the differences are summarized here:

Use IText:

- if you need to store styled text
- if you want to store international (non-ASCII) text
- if you want powerful language-sensitive comparison
- if you want to store really long runs of text

Note that IText has:

- no numeric support
- ties to the IOC Localization and 2DGraphics Frameworks

Use IString:

- if you need numeric formatting and parsing
- if the object is going to be used with a lot of char*-based system calls, which would force IText to create unnecessary overhead by transcoding repeatedly

Note that IString:

- uses styles via changing fonts
- is used by User Interface Classes

Text Display

Unicode Text

The Text and International Frameworks use the Unicode-based class IText as the storage mechanism for underlying text. The text can contain a mixture of runs of different character- and paragraph-styling information.

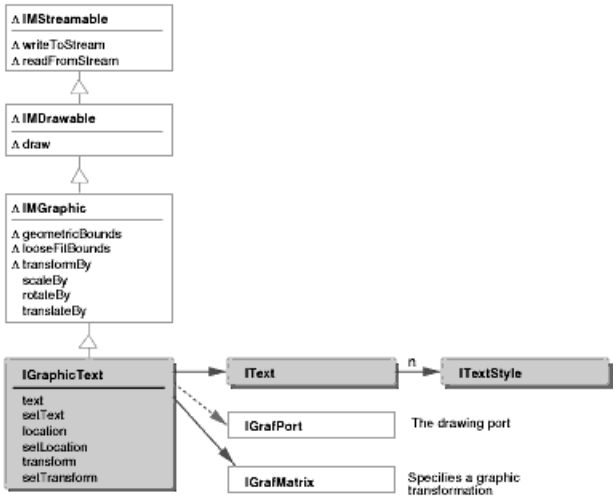
IGraphicText lets you display a single line of styled text as a static graphic object.

Static Text

The most straightforward mechanism for drawing the styled text represented in an instance of IText is provided by IGraphicText. IGraphicText implements functions inherited from IMGraphic for drawing, performing graphic transformations, and computing the bounds of the graphic in a device-specific manner.

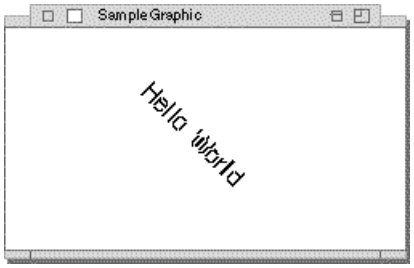
IGraphicText also provides access functions for the text and for the current transform.

This figure shows the IGraphicText interface:



After initial construction, IGraphicText lets you use setText to reset the text or the setTransform function to reset the current transform. You can also use the convenience function setLocation to set a new drawing position. This resets the current transform to translate the origin to the specified location. The containing view should call the draw function to redraw the string.

You can use the bounds returned by looseFitBounds to position the graphic. For example, you can create a matrix that rotates the graphic by 45 degrees around the center of the bounding rectangle:

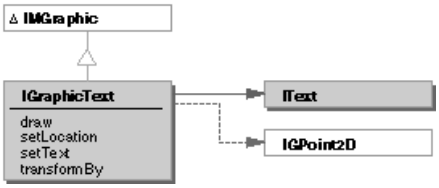


For computing the bounding rectangle, the function looseFitBounds returns more accurate results than geometricBounds because geometricBounds returns an approximate bounds based on an untransformed root screen port.

Some graphic transformations may also cause the bounding rectangle to change slightly. You should apply the transformation before computing and using the bounds.

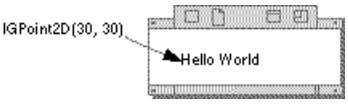
IGraphicText is a subclass of the 2D Graphics class IMGraphic that you use to draw static text strings. IGraphicText draws the styled text in an IText object as a single line of text, beginning at a specified point. IGraphicText also supports basic graphic transformations.

This figure shows IGraphicText and related classes:



IGraphicText is not intended to be subclassed. To implement your own static text display class, derive directly from classes in the 2D Graphics Framework.

You can specify an IGPoint2D object to represent the anchor position at which to draw the text. For English text, the origin is at the left baseline, as illustrated in the following figure:



To draw the text, you specify the IGrafPort into which to display the text. For example, this code demonstrates simple usage of IGraphicText:

```
void drawSomeText (const IText& styledText,
   IGPoint2D anchorLocation,
    IGrafPort* displayPort)
{
IGraphicText graphicText(styledText, IGraphicText::kSingleLine);
graphicText.setLocation(anchorLocation);
graphicText.draw(*displayPort);
}
```

Summary of Text Display Classes

The text display classes consist of the following classes:

Class	Description
IGraphicText	Lets you draw styled Unicode text strings as static graphics

Chapter 8. Styles

Styles and Style Sets

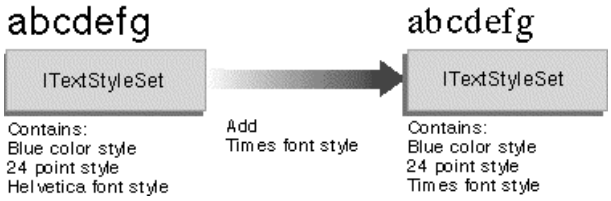
IText uses the classes ITextStyle and ITextStyleSet to encapsulate the styling information associated with a string. ITextStyle is the abstract base class for all text styles provided by the framework. ITextStyleSet provides a mechanism for manipulating styles in groups. Most functions that take styles as parameters allow you to add styles to a range of characters in an IText object either individually (with ITextStyle) or as a group (with ITextStyleSet).



A style object is basically a wrapper for some value that gives the style meaning. The type of value differs for each style. For example, a typeface style encapsulates a string that indicates which typeface to use, such as Times or Helvetica, while the *italic* style encapsulates a boolean that indicates whether the associated characters are *italicized* or not. You can change the value of a style object at any time.

Styles can define information that controls how the text is displayed—for example, the typeface or point size—or some other metadata not related to the display of the text, such as the natural language of the text or whether it is editable. However, styles do not define display mechanisms. It is up to the rendering mechanism to interpret the information in the style objects and display the text according to the associated styling information.

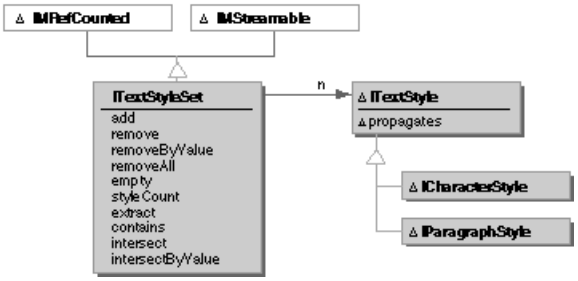
IText uses ITextStyleSet to store the set of styles associated with a particular style run, and you can use it to work with groups of styles. An ITextStyleSet object can contain only one instance of any given style. It cannot, for example, contain two instances of ITextTypefaceStyle even if one style indicates 'Helvetica' and another indicates 'Times.' If you add a style to a style set that already contains an instance of that type of style, the new style replaces the old style regardless of their values, as illustrated in this figure:



Style Classes

This figure shows the basic architecture for the Unicode Text Framework style classes:

© Copyright IBM Corp. 1998

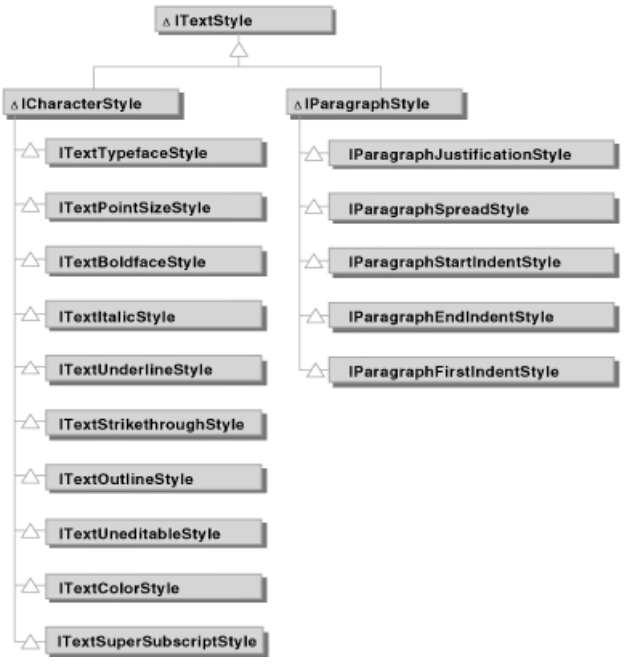


The framework currently supports two types of styles: character styles and paragraph styles. These styles derive from the abstract classes ICharacterStyle and IParagraphStyle, respectively, and are distinguished by how they *propagate*:

- Styles that propagate by character can be applied to any arbitrary range of characters, and maintain their association with those characters regardless of how the surrounding text changes. Character styles include point size and color.
- Styles that propagate by paragraph can be applied only to whole paragraphs—ranges of characters delimited by paragraph separator characters.

Paragraph styles include justification and indentation. For example, if you apply a paragraph style such as a justification style to a range of characters, IText automatically applies it to the entire paragraph (or multiple paragraphs) containing that range. Paragraph styles may be adjusted as text is edited, for example, if paragraph separator characters are added or deleted.

The framework includes fifteen concrete style classes:

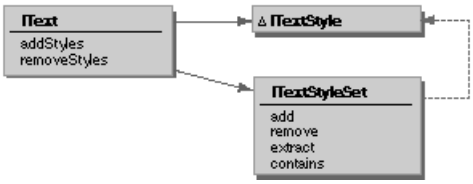


When you create a style object, give it a meaningful value. For example, you might give the value '12' to an ITextPointSizeStyle object or the value IParagraphJustificationStyle**::**kCenter to an IParagraphJustificationStyle object. No default values are provided—if you don't specify a value when you create a style object, the style's value remains undefined.

Using an undefined style may be useful, for example, in situations where the value is unimportant (such as calling IText::removeStyles) or where you create the style object to pass it to another function that will fill in the value.

The style classes also provide the ITextStyleSet, which lets you group styles. For example, you could add several styles to a range of characters with a single call to IText. Any IText function that takes an ITextStyle parameter is overloaded to take an ITextStyleSet parameter as well.

This figure shows the methods for setting styles on IText:



Character Style Classes

Character styles derive from ICharacterStyle. ICharacterStyle::propagates returns the value ITextStyle::kPropagateByCharacter. You can apply these character styles to any arbitrary range of characters, from a single character to all characters in an IText object.

This table describes the concrete character styles currently provided by the framework, along with the type of value each contains.

Class	Description	Value
ITextTypefaceStyle	Specifies the typeface to use to display characters.	An IText object encapsulating the name of the typeface, for example, 'Helvetica.'
ITextPointSizeStyle	Specifies the point size in which to display characters.	A `float` representing the point size, for example, '14.'
ITextBoldfaceStyle	Specifies whether characters are displayed in boldface, for example, SampleString.	A `bool-true` means the characters are displayed in boldface.
ITextItalicStyle	Specifies whether characters are displayed in italics, for example, *SampleString*.	A `bool-true` means the characters are displayed in italics.
ITextUnderlineStyle	Specifies whether characters are displayed underlined, for example, SampleString.	A `bool-true` means the characters are underlined.
ITextStrikethroughStyle	Specifies whether characters are displayed with a strikethrough line, for example, SampleString.	A `bool-true` means the characters are struckthrough.
ITextOutlineStyle	Specifies whether character are displayed in an outline typeface, for example, SampleString.	A `bool-true` means the characters are outlined.

Class	Description	Value
ITextUneditableStyle	Specifies whether characters can be edited. This is currently the only style that is not related to text display.	A `bool-true` means the character cannot be edited.
ITextColorStyle	Specifies a color to use to display characters.	Three `unsigned char` values representing the red, green, and blue components of the color.
ITextSuperSubscriptStyle	Specifies a shift from the baseline, in points, for displaying characters as superscripts or subscripts.	A `float` representing the baseline shift. A negative value creates a subscript. A positive value create a superscript.

Paragraph Style Classes

Paragraph styles derive from IParagraphStyle. IParagraphStyle::propagates returns the value ITextStyle::kPropagateByParagraph. You can apply paragraph styles only to whole paragraphs, that is, to ranges of characters delimited by UUnicodeSpecial::kParagraphSeparator characters.

If you apply paragraph styles to a range of characters that is less than a paragraph or that intersects multiple paragraphs, IText propagates the paragraph styles to all the relevant paragraphs. The styles also remain associated with the paragraph as the number of characters in the paragraphs changes during editing.

This table describes the concrete paragraph styles currently provided by the framework, along with the type of value each contains.

Class	Description	Value
IParagraphJustificationStyle	Specifies the alignment of the paragraph. Alignment is specified relative to the directionality of the writing system. For example, starting-edge-aligned means on the left for left-to-right languages such as English.	EJustification enum, defining the possible values kStart, kCenter, and kEnd.
IParagraphSpreadStyle	Specifies whether no lines, all lines, or all but the last line of the paragraph are fully justified.	ESpread enum, defining the possible values kNone, kAll, and kNotLastLine.
IParagraphStartIndentStyle	Specifies an indent distance, in points, for the starting edge (the left edge in left-to-right writing systems) of the paragraph.	A `float` representing the indent distance in points.
IParagraphEndIndentStyle	Specifies an indent distance, in points, for the ending edge (the right edge in left-to-right writing systems) of the paragraph.	A `float` representing the indent distance in points.

Class	Description	Value
IParagraphFirstIndentStyle	Specifies an indent distance, in points, for the starting edge of the first line of the paragraph.	A `float` representing the indent distance in points.

Currently the set of styles supported by the Unicode Text Framework is not extensible—that is, you cannot create your own subclasses of ITextStyle, ICharacterStyle, or IParagraphStyle. Future Open Class releases may support extending this style mechanism.

Style Propagation

The Unicode Text Framework uses the propagation of a style to determine how it can be applied to character ranges and how the affected style runs change as the characters are edited. The framework currently defines two types of style propagation:

- *Character-based* propagation
- *Paragraph-based* propagation

Styles that propagate by character can be applied to any arbitrary range of at least one character. The styles remain with these characters regardless of how characters within that range or surrounding that range are edited. Styles applied to a range of characters also propagate to any new text entered into that range.

For example, this shows a sequence of possible editing changes:

Enter text:

These are the times that try mens' souls.

Apply boldface and a new point size to a range:

These are the **times** that try mens' souls.

Edit text surrounding the new style run:

These are **times** that try mens' souls.

Edit text within the style run:

These are **ties** that try mens' souls.

Enter text into the style run:

These are **tales** that try mens' souls.

Styles that propagate by paragraph can be applied only to paragraphs—ranges of characters delimited by paragraph-separator characters (UUnicodeSpecial::kParagraphSeparator). This makes the style propagation mechanism more complicated than it is for character styles, because IText has to repropagate paragraph styles when paragraph separator characters are added or deleted.

The mechanism for propagating styles follows these rules:

- A text string containing no paragraph-separator characters takes on the paragraph styles of the text into which it is inserted.
- A text string bounded on both sides by paragraph-separator characters retains its paragraph styles even when inserted into text containing other paragraph styles.
- A range of text that is inserted into another string and then deleted should have the same paragraph separators and paragraph styles as it did before the editing operations.

IText propagates paragraph styles backwards from the character closest to a paragraph separator back to the beginning of the paragraph. For example, if you have two text strings containing several paragraphs with different paragraph styles:

String 1 with a single paragraph style run:

AAAAAAAAA¶

String 2 with 3 paragraph style runs:

BBBB¶CCCC¶DDDD

Insert string 2 into string 1:

AAAAABBBB¶CCCC¶DDDDAAAA¶

IText propagates paragraph styles yielding:

BBBBBBBBB¶CCCC¶AAAAAAAA¶

When propagating paragraph styles, IText recognizes only the UUnicodeSpecial::kParagraphSeparator character (U+2029) to be a paragraph-separator character. When text in other character encoding systems is transcoded into Unicode, you must ensure that it is processed by an ILineBreakConverter object so that the separator characters are transcoded correctly.

Character Data

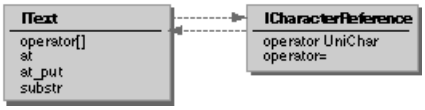
IText provides several functions for accessing specific characters or character ranges.

The framework provides a special class, ICharacterReference, that acts as a simple proxy class to access single `UniChar` values. IText uses ICharacterReference wherever a non-`const` reference to a `UniChar` is required. ICharacterReference provides a safer character reference mechanism by:

- Filtering out-of-range character references that might cause memory problems
- Protecting the integrity of an IText object's underlying storage mechanism, which might be shared
- Protecting against style propagation problems

ICharacterReference provides an operator that returns a copy of the referenced `UniChar` values.

This figure shows the low-level IText functions for accessing characters:



- operator[] and at return a reference to or copy of the character at a specified offset.
- at_put lets you set the character at a particular offset to a specified character. The ICharacterReference assignment operator also supports this.
- substr provides an IText object containing a copy of a specified range of characters.

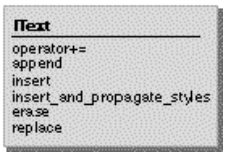
Editing Character Data

IText provides functions for editing character data. You can specify the text to add to the object as:

- Another IText object
- A `UniChar` array (null-terminated or with a specified length)
- A range of an IText defined by a pair of iterators
- A range of an IText defined by an offset and a character count

Unless noted, text added to an IText retains its character styling information (Unicode characters inserted into an IText are considered unstyled). Paragraph styles are propagated as appropriate.

This figure shows the IText interfaces for editing character data:



- operator+= and append add a string to the end of the current text.
- insert inserts a string at a specified offset (you specify the offset of the character after which new text is to be inserted).
- insert_and_propagate_styles inserts a string at a specified offset. The inserted string takes on the styles of the surrounding text.
- erase deletes the characters in a specified range.
- replace replaces a range of characters with another string.

Unlike IString::change, the IText::replace function is not able to perform a full search and replace.

Text and Style Run Iteration

Unicode Text Framework provides a set of iterators for accessing the character data in an IText object. These iterators all have the properties of a random-access iterator as described by most recent version of the C++ standard.

Character Iteration

The framework provides five iterators for iterating through the characters in an IText object. These iterators are compatible with the iterator classes provided by the C++ Standard Library:

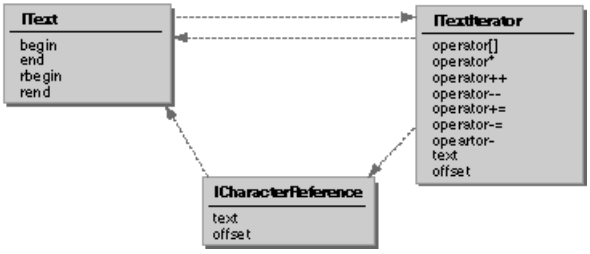
- ITextIterator provides iteration through the characters in an IText object.
- IReverseTextIterator provides backwards iteration through the characters in an IText object.
- IConstTextIterator is a variation on ITextIterator that doesn't allow you to modify the characters.
- IReverseConstTextIterator is a variation on IReverseTextIterator that doesn't allow you to modify the characters.

- IFastTextIterator provides faster iteration than the iterators listed above but fewer safety assurances. For example, the iterator is not guaranteed to stay valid if the underlying text is modified.

These iterators all follow the protocol for random-access iterators. You can start the iteration at any position in the IText object and iterate forward, backward, and so on. With the exception of IFastTextIterator, you should not construct iterators directly, but use the IText iterator functions begin, end, rbegin, and rend.

The iterators use the class ICharacterReference to return non-`const` references to `UniChar` values. This ensures that both the reference-counting mechanism used for IText storage and the paragraph style propagation are not disturbed.

This figure shows the basic protocol for the text iterators. Only ITextIterator is shown:



If you want to use IFastTextIterator, construct it from another text iterator. IFastTextIterator is different than the other text iterators in that it doesn't consider the styling mechanism or the shared storage mechanism. You can use IFastTextIterator when you want faster iteration. However, if you want to modify characters while using an IFastTextIterator, you need to ensure that the IText you are iterating over does not share storage with another IText. When using an IFastTextIterator:

- Do not initialize another IText object from the IText under iteration or assign the IText to another IText while the IFastTextIterator is in effect.
- Do not call non-`const` functions on the IText under iteration while the IFastTextIterator is in effect.
- Keep in mind that IFastTextIterator ignores styles when iterating over styled text. Paragraph styles will not be repropagated, and character styles will not be manipulated with their associated characters.
- Set the willWrite parameter to `false` if you want to use the iterator for read-only access of characters.

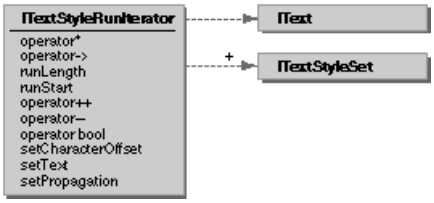
Style Run Iteration

The framework provides an additional iterator, ITextStyleRunIterator, that lets you iterate over the style runs in a styled IText object. A style run is a range of characters with identical styling information—that is, the same set of style objects with the same values. For example, the text string shown here has four style runs:



When using ITextStyleRunIterator, you can specify whether to iterate over character styles, paragraph styles, or both. The iterator considers both types of styles by default.

This figure shows the ITextStyleRunIterator interface:



- operator* and operator-> return the set of styles on the current style run. operator* returns a reference to the style set, while operator-> provides direct access to the members of the style set.
- runLength and runStart return the length and starting offset of the current style run.
- operator++ and operator— advance to the next or previous style set.
- operator bool indicates when iteration is complete.
- setCharacterOffset points the iterator to the style run that contains the specified offset.

When you construct the iterator, it points to the first style run.

Chapter 9. Character and Text Queries

Overview of Unicode Support

Many Open Class classes use the Unicode character encoding standard to represent text data internally. Unicode, a fixed-width, 16-bit character encoding system, contains codes for every character in every major world script, along with a wide set of symbols, punctuation, and control characters. Because the Unicode system can store and access every character, regardless of its script or natural language, it lets you manipulate text more easily than in environments that require multiple code pages to support different character sets.

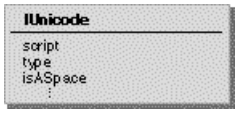
The Unicode support classes let you query the properties associated with individual Unicode character values. These properties, provided implicitly by the Unicode character encoding standard, include:

- information about the script (for example, Latin or Cyrillic)
- information about the character's character set (for example, symbols or control characters)
- semantic information, such as whether a character is a digit or is uppercase, lowercase, or uncased.

The IUnicode Class

The primary class in the Unicode support classes is IUnicode, which lets you determine a character's script and character properties. The Unicode support classes also provide a mechanism for referencing specific Unicode character values by name instead of by codepoint values.

IUnicode provides a set of static functions that check a Unicode character, represented by the datatype `UniChar`, for a specific property—for example, querying whether a character is an uppercase character, a digit, or one of the space characters. These functions let you check a character for a specific property without requiring you to know all the possibilities. You can test for a space character, for example, with the IUnicode::IsASpace function without needing to know the full set of Unicode characters used to represent a space.



The class library also provides a set of classes that contain enumerated names for each Unicode character value. These classes correspond to groups of characters based on script or functions: ULatin, UGreek, UDingbats, UMathematicalOperators, and so on. Use the names enumerated in these classes to reference specific Unicode character values.

Character Support

The Unicode character set provides full character coverage for the major scripts listed below, as well as for punctuation, symbols, and control characters. The character set for each script is independent—even if a character appears in multiple scripts, it has a separate code within each script. For example, the character *A* has

© Copyright IBM Corp. 1998

one code for the Roman alphabet, another code for the Greek alphabet, and yet another code for the Cyrillic alphabet. However, because more than one language may use a given alphabet, the character *A* is represented by the same code for English, French, and, in fact, all languages that use the Roman alphabet.

Supported Scripts

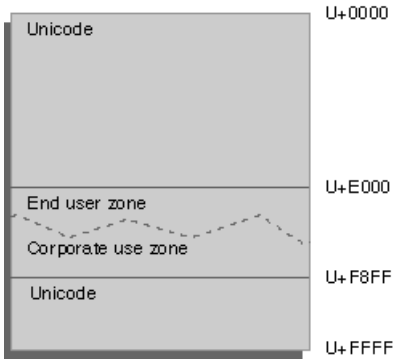
Arabic	Georgian	Hangul	Malayam	Thai
Armenian	Greek	Hebrew	Oriya	Zhuyinfuhao
Bengali	Gujarati	Kana	Roman	
Cyrillic	Gurmukhi	Kannada	Tamil	
Devanagari	Han	Lao	Telugu	

Reserved Areas

The Unicode standard sets aside a range of characters—from U+E000 to U+F8FF—for private use for:

- special characters or sets of characters not included in the Unicode set
- assigning specific semantics to a character

By convention, this area is divided into an end-user zone, which begins at U+E000 and ascends toward higher niumbers, and a corporate use zone, which begins at U+F8FF and descends toward lower numbers. The purpose of this convention is to minimize conflicting assignments within the private use area.



Character Values

The class library provides a name, through a set of enumerations, for every character in the Unicode set, with the exception of most of the Han ideographic characters. Names are provided for some particularly significant ideographs, such as digits and the 214 KangXi radicals.

To refer to specific Unicode values, use character names rather than code points. For example, refer to UGeneralPunctuation::kQuestionMark rather than the value U+003F.

Because of the large number of characters, the names are scoped into a set of classes based on script or function. These classes are provided only for referencing

the enumerated names they contain—do not use them for any other reason. These classes are listed in the following table. See the enumeration in the referenced header file for specific character names.

Character Name Enumerations

Category	Header file	Classes
General utility characters	iugnrl.hpp	UASCII UCombining UControlCode UGeneralPunctuation ULatin ULatin1 UModifierLetter UUnicodeDigit UUnicodeSpecial
East Asian scripts	iueasia.hpp	UBopomofo UHangulChoseong UHangulJongseong UHangulJungseong UHangulLetter UHangzhouNumeral UHanNumeral UHiragana UIdeographicAnnotation UKangXiRadical UKatakana UKatakanaHiragana
South and Southeast Asian scripts	iusasia.hpp	UBengali UDevanagari UGujarati UGurmukhi UKannada ULao UMalayalam UOriya UTamil UTelugu UThai
Eastern European scripts	iueeuro.hpp	UCoptic UCyrillic UGeorgian UGreek
Mideastern scripts	iumeast.hpp	UArabic UArmenian UHebrew

Category	Header file	Classes
Characters provided for compatibility with other standards	iucmpbty.hpp	UArabicCompatibility UArabicLigature UArmenianSmallLigature UBlocks UBoxDrawings UCircledDigit UCircledHangul UCircledIdeograph UCircledKatakana UCircledLatin UCircledNumber UCJKCompatibility UCJKSquaredAbbreviations UCJKSquaredWords UCNSCompatibility UFullStopDigit UFullStopNumber UFullwidth UHalfwidth UHalfwidthHangulLetter UHangulSyllable UIdeographicTelegraph ULatinSmallLigature UParenthesizedDigit UParenthesizedHangul UParenthesizedIdeograph UParenthesizedLatin UParenthesizedNumber UPresentationFormForVertical URomanNumeral USmallVariants USubscript USuperscript UVulgarFraction
Symbols	iusyms.hpp	UAPLFunctionalSymbol UArrow UCJKSymbols UControlCodePicture UCurrency UDingbats UGeometricShapes UHarpoon ULetterLikeSymbol UMathematicalOperators UMiscellaneousTechnical UOCR UStandardPhonetic UZapfDingbats

Some character names may refer to characters of types other than `UniChar`. In such cases, you may need to cast characters before using interfaces that take `UniChar` parameters.

Character Properties

IUnicode provides static member functions that let you access the semantic information provided by the Unicode character standard. These functions let you determine the script of a `UniChar` character or query whether that character has a particular property.

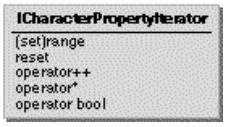
IUnicode includes an enum, EUnicodeScript, that defines the set of scripts supported by Unicode. The function script returns the correct enumerated value that indicates the script of the character. Other functions return a boolean value that indicates whether the character has a particular property.

This figure shows the IUnicode interface:



ICharacterPropertyIterator lets you scan the set of Unicode characters for characters that have a specific set of properties. For example, you might use this class to return a list of punctuation characters for a particular script. The iterator takes a range of character properties, defined by the IUnicode enum ECharacterProperty, and identifies the characters that have the properties in that range.

This figure shows the ICharacterPropertyIterator interface:



- operator++ lets you advance the iterator
- operator* lets you access the character currently referenced by the iterator
- operator bool indicates when the iterator has reached the last character with the specified properties

Chapter 10. Streams and Storage

Introduction to the File System

The VisualAge C++ File System Framework gives you access to all file system objects such as volumes, directories, files, file contents and operations. Its interface provides classes that represent physical file system objects and allows you to manipulate those objects.

Some typical tasks you can perform using the File System Framework are:

- Creating, moving, copying, deleting, locating, and otherwise manipulating file system objects
- Manipulating and parsing pathnames in a portable manner
- Accessing the attributes of files, directories, and volumes
- Accessing the contents of volume, directory, and file objects
- Iterating through directories and volumes

File System Framework functions are organized into six broad categories of classes:

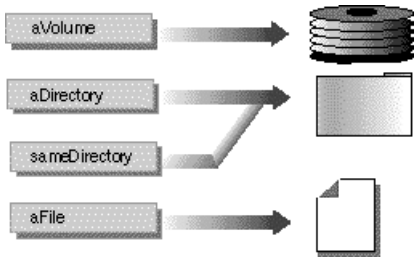
- File System Entities
- Pathnames and Pathname Parsers
- Data Accessors
- File System Iterators
- File System Movers and Copiers
- File System Exceptions

Overview of File System Entities

The File System Framework enables you to use surrogate instances called file system entities to access physical objects that reside on hardware or virtual devices. These surrogate objects are instances of IFileSystemEntity and its subclasses: IFile, IDirectory and IVolume.

- IFileSystemEntity is the base class for all file system entities.
- IFile provides a surrogate object for a file.
- IDirectory provides a surrogate object for a file system directory.
- IVolume provides a surrogate object for a file system volume.

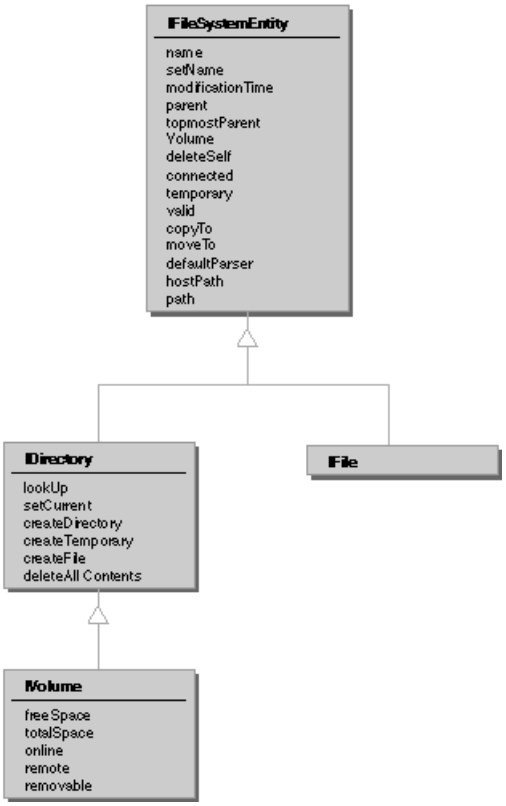
The following figure shows the relationship of surrogate instances to physical objects:



Multiple surrogate objects can refer to the same physical object.

© Copyright IBM Corp. 1998

The IFileSystemEntity class defines the common protocol for file system entities. IFile, IDirectory and IVolume derive from IFileSystemEntity and implement protocols specific to each kind of entity.



Files

An IFile instance is a surrogate entity that maps to a physical file. You locate and create IFile instances using the IFile constructors and IDirectory's lookUp and createFile methods. You can also delete the physical file entity by using a deleteSelf method.

Directories

An IDirectory instance is a surrogate entity that maps to a physical directory. With it, you can look up an entity within a directory, create a new directory, create a new file, delete the contents of a directory, and delete the directory itself.

Volumes

IVolume derives from IDirectory because it represents the root directory of a volume. You can treat an IVolume instance much as you would treat an IDirectory instance. IVolume also has several methods you can use to access attributes that are specific to volumes.

Attributes

Types of attributes for a file system object include:

- Name
- Creation date and time
- Modification date and time

In addition, IFile has a size attribute which you access using the size and setSize functions.

IFileSystemEntity implements functions for accessing the attributes that are common to all file system objects. IDirectory, IVolume, and IFile also implement accessors for attributes that are specific to their corresponding physical file system objects. For example, IFile instances manage end-of-file information stored as an attribute. They also manage storage of the file name, creation date, modification date, file size, and entity type (which in this case is always file).

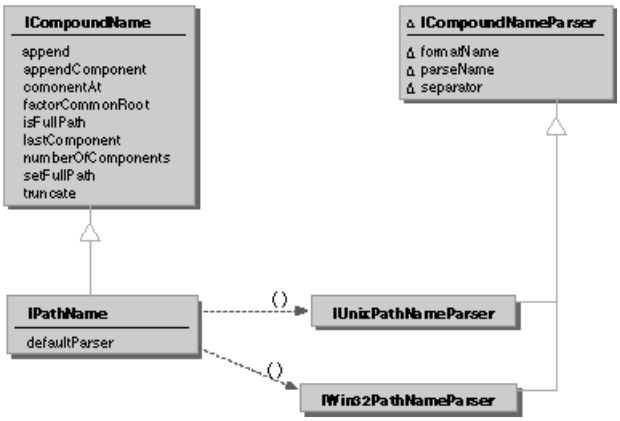
Overview of Pathnames and Pathname Parsers

A pathname specifies the physical location of an object on a mounted volume. The File System Framework provides classes which enable you to specify pathnames in a portable manner, or in a host-specific way. You use pathnames to:

- Create a file system entity
- Retrieve a file system entity based on a pathname
- Get a pathname based on the file system entity
- Store and manipulate pathnames across platforms in a portable manner

The classes associated with pathnames and pathname parsers are

- ICompoundName
- IPathName
- ICompoundNameParser
- IUnixPathNameParser
- IWin32PathNameParser



You can obtain a file system entity by providing the pathname of the entity you want, and you can get the pathname of an entity by providing the file system entity itself. The file system gives you two ways to use pathnames with file system entities:

- You can store pathnames as strings in the format used by a particular operating system.
- You can store and manipulate pathnames in a portable manner.

Host-Specific Pathnames

Host-specific paths are simple strings that store pathnames in the format used by a particular operating system. Some examples of host-specific pathnames are:

UNIX	/tmp/file.txt files/new/readme.txt
Windows and OS/2	C:\temp\file.txt files\new\readme.txt

Pathname Parsers

Host-specific pathnames are not portable. For example, if you try to use a Windows-style pathname on a UNIX system, you will get an exception. The File System Framework provides a way to avoid this problem. All methods that take a host-specific path as a parameter also take an optional ICompoundNameParser parameter, which you can use to specify a parser for the pathname. If you do not explicitly specify a parser, you will get the parser that matches the path format of the host operating system:

The pathname parsers provided in this release of Open Class are

- IUnixPathNameParser—a concrete class that parses and formats UNIX-style file system pathnames
- IWin32PathNameParser—a concrete class that parses and formats Windows-style file system pathnames

Portable Pathnames

Although host-specific pathnames are easy to use, you sometimes need to store or manipulate a pathname in a portable manner. For example, a text document might need to include the relative path to an embedded graphic file. The class IPathName provides this portable functionality.

IPathName, derived from ICompoundName, represents full or partial paths to file system entities. It allows you to specify file system entities via their absolute or relative pathnames.

You can also construct an IPathName from a IDirectory instance and a partial path relative to that directory. The partial path can be specified either as an IHostPathName or as another IPathName.

Data Accessors

Data access is provided through the Open Class streaming classes.



IFileIOStream is a standard library-style stream class that can be constructed from an IFile. It allows iostream-based code to interoperate with the file system API. To perform I/O operations, you use the functions inherited from iostream. Use IFileIOStream when you need compatibility with the standard iostream classes or access to legacy data that is stored in files. You use this class almost exactly the same way you would the standard fstream class, except that the constructor accepts an IFile argument.

IDataStream is the standard Open Class stream for C++ objects. Use IDataStream to store persistent objects, Open Class documents, and other object-based data.

The constructors for both of the stream classes take two int arguments, modeForMe and modeForOthers, that determine how the file is opened. The first argument specifies the type of access that you want to have to the stream yourself, plus any special behavior that you want when the file is opened. The second argument specifies the access that you want to allow for others who try to open the file when you already have it open.

The int values for both of these arguments are constructed by combining the open_mode constants from the ios class or the EOpenMode constants from the IFile class. The constants that specify access levels are as follows:

- 0—allows no access to the file. This is the default value of the modeForOthers argument.
- in—allows read access. The file can be read but not written to.
- out—allows writing to the file.

There are a few other open-mode constants that specify special behavior when you open or write to the file. These constants have no effect if they are added into the modeForOthers argument.

- app—tells the stream to seek to the end of the file before all write operations.
- ate—causes a seek to the end immediately after the open, but not before each write.
- binary—performs input and output in binary, as opposed to text, mode.
- trunc—truncates an existing stream when opening it.

Both IDataStream and IFileIOStream keep track of the end of the file, also called end of stream. As you write out a file stream, the end-of-file marker moves forward. To move the stream marker more efficiently, use IFileIOStream::seek to move the marker forward to the position you want.

Note: When you move the end-of-file marker forward, the results will depend on the host file system. Because your code might be used on different file systems, you need to code as though you risk running out of file space each time you move the end-of-file marker.

File System Iterators

The File System Framework provides three classes that you can use for iterating through directories and volumes:

- IDirectoryIterator iterates over the files and directories contained within a single directory or an entire directory hierarchy.
- IRootDirectoryIterator iterates over all topmost directories on all mounted volumes.
- IVolumeIterator iterates over all the local disks and network volumes that are currently mounted.

OS/2 WIN IVolumeIterator returns an IVolume for each disk connected to the system and for each connected network drive. Additionally, because each volume is represented as a root directory, IRootDirectoryIterator will return the same entities as IVolumeIterator.

AIX There is only one root directory ('/') on UNIX systems.

IFileSystemEntity's moveTo and copyTo Functions

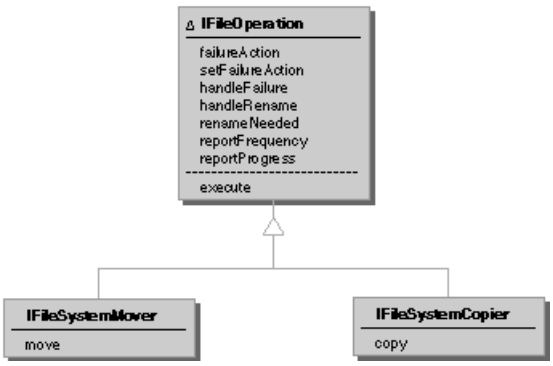
The IFileSystemEntity class provides functions that copy or move a file or directory into a specified destination directory:

- The copyTo function copies an entity from its current location to another location
- The moveTo function moves an entity from its current location to another location

These functions are not meant to be customized.

Mover and Copier Classes

The File System Framework provides the following classes which you can subclass if you need customized copy or move behavior:

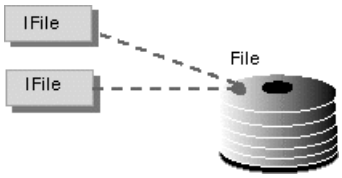


- IFileOperation is an abstract base class for classes that operate recursively over a tree of file system entities. It is the parent of IFileSystemMover and IFileSystemCopier.
- IFileSystemMover moves files and directories from one IDirectory instance to another, recursively moving all of an entity's children.
- IFileSystemCopier copies files and directories between IDirectory instances, recursively copying all of an entity's children.

These classes provide the infrastructure for progress indicators, custom failure handling, and resolution of file-naming conflicts. You can instantiate and use IFileSystemMover and IFileSystemCopier directly, or subclass them in order to implement progress reporting, name-conflict resolution and failure handling. You can derive from IFileSystemCopier or IFileSystemMover to handle failures, resolve name conflicts, or report progress. You can also create an entirely new file operation by subclassing from IFileOperation. For example, you could create a subclass of IFileOperation that adds up the total size of all files in a directory tree, or searches through all of the files for a particular piece of data.

Thread Safety

A surrogate instance (a file system entity) is only a handle to a physical entity. Multiple file system entities can point to a single physical entity.



The classes that comprise the File System interface are not multithread-safe. You cannot share individual file system entity instances. Even though the entity instances themselves are not thread-safe, however, the underlying physical objects they represent usually are. You can make copies of an instance and simultaneously use the original and each copy in a different thread.

Copies of file system instances are very inexpensive to construct, and the File System interface is designed to ensure that the implementation of copy semantics is efficient.

Note: If the files themselves are not thread-safe—for example, if there are two unsynchronized threads writing to a file at once—making copies of the entity instances will not help.

File System Exceptions

All file system exceptions derive directly or indirectly from IException. All of the standard IException behavior will work with these classes.

- IEntityInUse indicates an attempt to move, delete, or otherwise manipulate an entity that is currently in use.
- IEntityInvalid indicates an attempt to perform an operation on a file system entity that does not point to a valid on-disk entity.
- IEntityTypeMismatch indicates an attempt to assign the wrong type of file system entity to another entity subclass.
- IFileOperationCancelled indicates that the user canceled a file system operation.
- IVolumeOffline indicates an attempt to perform an operation on a volume that is off line.

Chapter 11. Streaming Classes

Introduction to the Streaming Classes

The VisualAge C++ streaming classes provide support for the storage and retrieval of data, including both C++ objects and primitive types, from streams. The Streaming Classes comprise three core elements:

- Data streams
- The IMStreamable mixin class
- Stream modules

The Streaming Classes also define several exception classes.

Capabilities supported by the Streaming Classes include monomorphic streaming, polymorphic streaming, platform-to-platform data compatibility (a document streamed out on one platform will be readable on another platform), and release-to-release data compatibility (a document written by one version of an application or system will be readable by a newer or an older version).

Typical usage of the Streaming Classes involves:

- Implementing persistent storage
- Marshalling function arguments for remote procedure-calls
- Streaming data polymorphically and monomorphically
- Implementing release-to-release data compatibility
- Streaming aliases and aliased objects

Exceptions Defined by the Streaming Classes

Streaming defines the following exception classes:

- IInvalidDataOnStream
- IUnknownTypeOnStream
- IEndOfStream
- IAddressAlreadyInContext
- IInvalidContextNumber

These exceptions can be raised while reading from a stream. They generally indicate that the stream data is corrupt or that the type of data written does not correspond to the type of data that the program is attempting to read from the stream.

Aliased Objects

Streaming supports the detection and correct handling of aliased pointers to objects.

Suppose that:

- A and B are two objects to stream out
- A and B each have a pointer to the same third object C
- Both A and B are designed to write C to the stream

© Copyright IBM Corp. 1998

- When A and B are streamed back in, you need to create one object C to which both A and B point

Without the detection of aliased objects, writing A and B to the stream would cause two copies of C to be written to the stream. Reading the streamed data back in would then result in the creation of two copies of C, one associated with A, the other with B. This would not only waste memory, but would also be logically incorrect. Changes made in one copy of C would not be reflected in the other, and thus would not be seen by both A and B.

With the appropriate use of writeAliasedObject instead of plain writeObject, the duplicate write of C would be recognized, and would be replaced by a flag, on the stream, that refers back to the initial write. When A and B are read, only one C would be created. The pointers in both A and B would refer to this single object.

Note: For some derived classes with multiple virtual inheritance, base classes may be reachable through multiple paths of the inheritance hierarchy. When a client streams a single instance of the derived class, the base class streaming functions may be invoked more than once. However, streaming classes ensure that there will no adverse effect on the results.

Monomorphic Streaming

Monomorphic streaming is used when you know the exact type of the data you are streaming. Compared to polymorphic streaming, this can save some overhead. Monomorphic streaming is useful for simple data types, and structs and objects without vtables.

Polymorphic Streaming

Polymorphic streaming is used to stream an object given a pointer or reference to one of its base classes. This is similar to calling a function which takes a base class pointer, but passing in a pointer to a derived type. When a client reads from the stream, objects of the correct type are automatically constructed and initialized from the stream data.

Consider, for example, the subclasses of IMGraphic. A typical drawing has lines, ellipses, and curves, as well as polygons. If you are streaming out many of these objects, you don't want to have to remember the precise sequence of data types that you have streamed out. Polymorphism lets you classify all these objects as an IMGraphic. You can then let the runtime type system figure out the precise class of the object being streamed in to a pointer of type IMGraphic.

Polymorphic streaming, however, incurs more overhead than monomorphic streaming in terms of time and space. Polymorphic streaming includes the type of each object along with the object's state in the external (stream) representation of the object. This extra information gives the runtime system the information it needs to reconstruct the correct object when it is streamed back in.

Release-to-Release Data Compatibility (RRDC)

Class version mismatches can occur when two machines running different software versions exchange data or, on a single machine, when upgraded software accesses older data. With VisualAge C++ streaming classes, however, objects that are created and written to a stream by one version of a class implementation are readable on either a newer or older version.

Release-to-release data compatibility does not make use of version numbers. Class data on a stream are bounded by a frame that allows stream in functions for a class to know when they have reached the end of the data that was written or to skip over any extra data. Thus, new implementations are able to sense that they are reading an older stream data, and do not try to read data that does not exist on the stream. Old class implementations are able to sense and skip over extra data that was written by newer versions.

RRDC can support any number of versions of a class. The main requirements are:

- Each newer version must make its additions to the stream data format at the end of the data written by the previous version.
- Data expected by older versions must continue to be written by newer versions. If the new version doesn't use the old data, then at stream-out time the new version must synthesize the old data or at least a suitable approximation of it to enable old versions to work when they stream in the data.

Note:RRDC is only available when using one of the tagged stream formats (see Instantiating a Data Stream for available stream formats). Raw binary stream formats omit the tags that delimit object data on a stream. They are suitable for caching data, streaming between two threads in the same process, and for streaming between two processes on the same machine when both processes are using the same version of a class. Raw binary stream formats are not suitable for persistent data or for streaming between systems that may have different versions of the same DLLs installed. The class wishing to provide RRDC can not enforce the use of a tagged stream format; it's up to whoever specifies the stream to choose an appropriate encoding. RRDC enabled classes (those containing IStreamIn/OutFrames) will stream their data correctly on non-tagged stream; it's only the ability to detect the end of a frame on reading that is lost.

Additional stream frames can be used within a class to delimit variable length data.

Data Streams

You get an IDataStream object when you create a stream. This object handles the encoding of data onto the stream, the decoding of data off of the stream, and the buffering and physical IO of the stream data. IDataStream is not intended to be subclassed, instantiated directly, or copied. Data streams are created using stream creation functions from other classes (for example, IFile::createStream).

You can write objects to streams in any order you choose, so long as you read corresponding objects in the same order.

You must know the type of an object that has been written to a stream in order to read it from the stream. When you stream in an object pointer, you must stream into a pointer that is the same type as the object on the stream or one of the object's base classes.

If you write an object monomorphically, you must read it monomorphically. If you write an object polymorphically, you must read it polymorphically. Monomorphic streaming writes out only the bits that represent the value of the streamed object, while polymorphic streaming writes out object information in addition to the object's value.

Application Data Interfaces

A class that supports streaming should provide, as part of its documentation, a description of its data format on a stream. This application data interface (ADI) will enable developers to create, without access to the original source code, programs that can read the streamed data.

The ADI should include the type (class or primitive) and purpose of each item written by the stream out function of the class, in the order that they are written. In the case of polymorphic streaming, the ADI need only specify a base class type.

An ADI for a class would not include a description of the encoding for primitive types or the stream tags and control information used by the IStreamIO class. It would also not include the description for any embedded classes; these would be obtained directly from the description of the embedded class.

Note: Applications which need to interoperate with other applications that are not built with VisualAge may want to support a data format that does not use Open Class streaming. For example, a word processor would probably provide a way to save or export a plain-text file for exchange with other applications.

Chapter 12. I/O Stream Classes

Overview of I/O Streams

The I/O Stream Classes provide the standard input and output capabilities for C++. In C++, input and output are described in terms of *streams*. The processing of these streams is done at two levels. The first level treats the data as sequences of characters; the second level treats it as a series of values of a particular type.

There are two primary base classes for the I/O Stream Classes:

1. The streambuf class and the classes derived from it (strstreambuf, stdiobuf, and filebuf) implement the *stream buffers.* Stream buffers act as temporary repositories for characters that are coming from the *ultimate producers* of input or are being sent to the *ultimate consumers* of output.
2. The ios class maintains formatting and error-state information for these streams. The classes derived from ios implement the formatting of these streams. This formatting involves converting sequences of characters from the stream buffer into values of a particular type and converting values of a particular type into their external display format.

The I/O Stream Classes predefine streams for standard input, standard output, and standard error. If you want to open your own streams for input or output, you must create an object of an appropriate I/O Streams class. The iostream constructor takes as an argument a pointer to a streambuf object. This object is associated with the device, file, or array of bytes in memory that is going to be the ultimate producer of input or the ultimate consumer of output.

Input and Output for User-Defined Classes

You can overload the input and output operators for the classes that you create yourself. Once you have overloaded the input and output operators for a class, you can perform input and output operations on objects of that class in the same way that you would perform input and output on char, int, double, and the other built-in types.

The I/O Stream Class Hierarchy

The I/O Stream Classes have two base classes, streambuf and ios.

The streambuf class implements *stream buffers*. streambuf is the base class for the following classes:

- strstreambuf
- stdiobuf
- filebuf

The ios class maintains formatting and error state information for streams. Streams are implemented as objects of the following classes that are derived from ios:

- stdiostream
- istream
- ostream

© Copyright IBM Corp. 1998

The classes that are derived from ios are themselves base classes.

istream is the input stream class. It implements stream buffer input, or *input* operations. The following classes are derived from istream:

- istrstream
- ifstream
- istream_withassign
- iostream

ostreamis the output stream class. It implements stream buffer output, or output operations. The following classes are derived from ostream:

- ostrsteam
- ofstream
- ostream_withassign
- iostream

iostream is the class that combines istream and ostream to implement input and output to stream buffers. The following classes are derived from iostream:

- strstream
- iostream_withassign
- fstream

Note: The I/O Stream Classes also define other classes, including fstreambase and strstreambase. These classes are meant for the internal use of the I/O Stream Classes. Do not use them directly.

I/O Stream Header Files

To use an I/O Stream class, you must include the appropriate header files for that class. The following lists the I/O Stream header files and the classes that they cover:

iostream.h contains declarations for the basic classes:

- streambuf
- ios
- istream
- istream_withassign
- ostream
- ostream_withassign
- iostream
- iostream_withassign

fstream.h contains declarations for the classes that deal with files:

- filebuf
- ifstream
- ofstream
- fstream

stdiostream.h contains declarations for stdiobuf and stdiostream, the classes that specialize streambuf and ios, respectively, to use the FILE structures defined in the C header file stdio.h. Note: the 8.3 file naming convention compliant name of this file on OS/2 is **stdiostr.h**.

strstream.hcontains declarations for the classes that deal with character strings. Note: the 8.3 file naming convention compliant name of this file on OS/2 is **strstrea.h**. The first 'str' in each of these names stands for 'string':

- istrstream
- ostrsteam
- strstream
- strstreambuf

iomanip.h contains declarations for the parameterized manipulators. Manipulators are values that you can insert into streams or extract from streams to affect or query the behavior of the streams.

stream.h is used for compatibility with earlier versions of the I/O Stream Classes. It includes iostream.h, fstream.h, stdiostream.h, andiomanip.h, along with some definitions needed for compatibility with the AT&T C++ Language System Release 1.2. Only use this header file with existing code; do not use it with new C++ code.

Note: If you use the obsolete function form() declared in stream.h, there is a limit to the size of the format specifier. If you call form() with a format specifier string longer than this limit, a runtime message will be generated and the program will terminate.

Stream Buffers

One of the most important concepts in the I/O Stream Classes is the stream buffer. The streambuf class implements some of the member functions that define stream buffers, but other specialized member functions are left to the classes that are derived from streambuf: strstreambuf, stdiobuf, and filebuf.

Note: The AT&T and UNIX System Laboratories C++ Language System documentation use the terms *reserve area* and *buffer* instead of *stream buffer*.

What Does a Stream Buffer Do?

A stream buffer acts as a buffer between the *ultimate producer* (the source of data) or *ultimate consumer* (the target of data) and the member functions of the classes derived from ios that format this raw data. The ultimate producer can be a file, a device, or an array of bytes in memory. The ultimate consumer can also be a file, a device, or an array of bytes in memory.

Why Use a Stream Buffer?

In most operating systems, a system call to read data from the ultimate producer or write it to the ultimate consumer is an expensive operation. If your applications can reduce the number of system calls they have to make, they will usually be more efficient. By acting as a buffer between the ultimate producer or ultimate consumer and the formatting functions, a stream buffer can reduce the number of system calls that are made.

Consider, for example, an application that is reading data from the ultimate producer. If there is no buffer, the application has to make a system call for each character that is read. However, if the application uses a stream buffer, system calls will only be made when the buffer is empty. Each system call will read enough characters from the ultimate producer (if they are available) to fill the buffer again.

Format State Flags

The ios class defines an enumeration of format state flags that you can use to affect the formatting of data in I/O streams. The following list shows the formatting features and the format flags that control them:

- Whitespace and padding: ios::skipws, ios::left, ios::right, ios::internal
- Base conversion: ios::dec, ios::hex, ios::oct, ios::showbase
- Integral formatting: ios::showpos
- Floating-point formatting: ios::fixed, ios::scientific, ios::showpoint
- Uppercase and lowercase: ios::uppercase
- Buffer flushing: ios::stdio, ios::unitbuf

Using Manipulators

Manipulators provide a convenient way of changing the characteristics of an input or output stream, using the same syntax that is used to insert or extract values. With manipulators, you can embed a function call in an expression that contains a series of insertions or extractions. Manipulators usually provide shortcuts for sequences of iostream library operations.

The iomanip.h header file contains a definition for a macro IOMANIPdeclare(). IOMANIPdeclare() takes a type name as an argument and creates a series of classes you can use to define manipulators for a given kind of stream. Calling the macro IOMANIPdeclare() with a type as an argument creates a series of classes that let you define manipulators for your own classes. If you call IOMANIPdeclare() with the same argument more than once in a file, you will get a syntax error.

Simple Manipulators and Parameterized Manipulators

There are two kinds of manipulators:

Simple manipulators do not take any arguments. The following classes have built-in simple manipulators:

- ios
- istream
- ostream

Parameterized manipulators require one or more arguments. setfill (near the bottom of the iomanip.h header file) is an example of a parameterized manipulator. You can create your own parameterized manipulators and your own simple manipulators.

ios Methods and Manipulators

For some of the format flags defined for the ios class, you can set or clear them using an ios function and a flag name, or by using a manipulator. With manipulators you can place the change to a stream's state within a list of outputs for that stream.

The I/O Stream Classes and stdio.h

In both C++ and C, input and output are described in terms of sequences of characters, or *streams*. The I/O Stream Classes provide the same facilities in C++ that stdio.h provides in C, but it also has the following advantages over stdio.h:

- The input or extraction (>>) operator and the output or insertion (<<) operator are typesafe. They are also easy to use.
- You can define input and output for your own types or classes by overloading the input and output operators. This gives you a uniform way of performing input and output for different types of data.
- The input and output operators are more efficient than scanf() and printf(), the analogous C functions defined in stdio.h.. Both scanf() and printf() take format strings as arguments, and these format strings have to be parsed at run time. This parsing can be time-consuming. The bindings for the I/O Stream output and input operators are performed at compile time, with no need for format strings. This can improve the readability of input and output in your programs, and potentially the performance as well.

Chapter 13. Windows, Menus, and Events

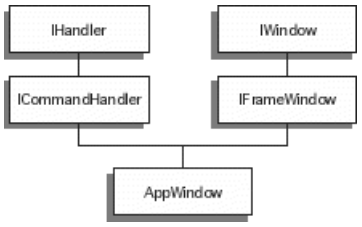
Windows

When you develop an application, you usually start with a window that is a composite of a frame window, title bar, menu bar, system menu, maximize and minimize buttons, client window, and extensions. The frame window coordinates the actions of the frame extensions and client window, enabling the composite window to act as a single unit. A frame's client window is the control that corresponds to the rectangular portion of the frame window not occupied by other controls (menus or buttons).

Frame Windows

A *frame window* is a window that an application uses as the base when constructing a main window or other composite window, such as a dialog window or message box. A frame window provides basic features, such as borders and a title bar. It can also have a set of resources associated with it, such as icons, menus, and accelerators.

The hierarchy shown below shows how classes are derived from parent classes. This is not a complete hierarchy because the parent classes for IHandler and IWindow are not shown. However, the diagram gives you an idea about how classes are derived for a basic window.



The *parent* window is used mainly for window location. For example, most frame windows have the desktop as a parent so that they can appear anywhere on the desktop. A *child* window is a window that is only visible inside of another window, the parent. The child is 'clipped' to its parent, meaning it is confined within it, and it cannot move beyond the boundaries of the parent window.

The *owner* window is mainly used for message passing. Several messages are passed up the owner chain for processing and changes to color and font are passed down the owner chain. A parent window must be an owner window.

OS/2 When you move an owner window, all of the windows that it owns are moved as well except if the IFrameWindow::noMoveWithOwner style is specified.

On all other environments, noMoveWithOwner is the default and cannot be changed.

Windows does not allow the owner of a top-level window to be modified after the window is created. Child window owners can be changed using the Open Class Class Library but not when using native Windows calls.

© Copyright IBM Corp. 1998

A *primary* window has the desktop as its parent and owner. A *secondary* window has the desktop as its parent, but its owner is another window. A child window would have another window for its parent and owner.

Note: A frame window can be a primary window, secondary window, or a child window.

Menus

A *menu* is a window that contains a list of items—text strings, bitmaps, or images drawn by an application—that enables the user to choose from these predetermined choices using a mouse or keyboard. The types of menus are the menu bar, pull-down menu, cascaded menu, pop-up menu, and system menu. A menu can be created dynamically using program functions, or it can loaded from a resource file. A menu is always owned by another window, usually a frame window.

A *menu item* is a representation of one of these predetermined choices. A menu item can represent either a command choice, or it can represent an item that when selected, brings up an additional list of choices to choose from. When a user selects a menu item which represents a command choice, a menu selection event is sent to a command handler.

Menu Bars and Pull-Down menus

The *menu bar* is the area near the top of a window, below the title bar and above the client area of the window. A menu bar is a horizontal menu that contains a list of choices. When a user selects a choice on a menu bar, a pull-down menu associated with that choice is displayed. A *pull-down* menu is a vertical menu that appears below the menu bar as the result of selecting a menu item that represents this list of choices. Typically, all of the menu items on a menu bar represent pull-down menus. In turn, a pull-down menu item can display another pull-down menu item, also called a *cascaded menu*. It is recommended by several GUI style guides that you use no more than three levels of cascaded menus or create a dialog when you have more than three. The pull-down menus ordinarily are hidden, but they become visible when the user makes a selection from the menu bar.

The User Interface Class Library provides the IMenu and IMenuBar classes to allow you to create menu bars, menu items, and pull-down menus or map those items from a menu loaded from a resource. The IMenu class is an abstract base class that provides the common behavior shared by all of the IMenu derived classes, including IMenuBar, IPopUpMenu, ISubMenu, and ISystemMenu. It contains member functions that can be used to manipulate menus according to the needs of your application. You can add, replace, remove, select, and disable menu items in the menu. This class also provides functions to access properties such as the following:

- The color of the menu
- The number of menu items in the menu
- The bounding rectangle for menu items in the menu

Objects of the IMenu (and derived) classes are not windows themselves. This is due to the implementation of menus in the Windows operating system as non-client areas of the frame. Application code, which relies on the fact that an IMenu is a window in the OS/2 operating system, may not be portable to the Windows operating system.

To support menus loaded from a resource file, IMenu classes have the following requirements:

- Menu resources must be defined using the PM MENU keyword or Windows MENUEX keyword.
- Unique IDs must be provided for all pop-up menu items, indicated by the POPUP keyword.

AIX

IMenu does not support conditional cascading behavior or the drawItem style (drawn buttons) in menus.

The User Interface Class Library provides a callback routine to process the following menu events:

- Showing a menu on the display
- Selecting a menu item in a menu
- Removing a menu from the display

Pop-Up Menus

A *pop-up menu* is a menu that is displayed when a user presses the appropriate key or mouse button. A pop-up menu contains choices that can be applied to an object at the time the menu is displayed.

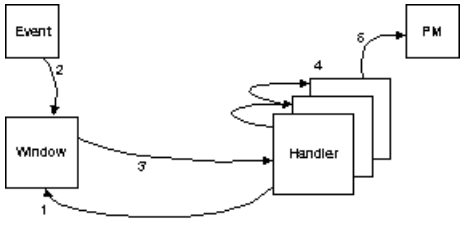
The User Interface Class Library provides the IPopUpMenu and IMenuHandler classes to manipulate pop-up menus. To display a pop-up menu in your application, derive your class from IMenuHandler, override the makePopUpMenu function, and construct an IPopUpMenu object.

Events and Event Handlers

The Open Class Class Library uses events and event handlers to encapsulate and hide the message architectures of Windows, OS/2 Presentation Manager (PM) and Motif in an object-oriented way. The Open Class Class Library reserves message IDs beginning at 0xFF00, IC_UM_BASE. If you use the Open Class Class Library, define user messages only in the range of WM_USER (0x1000) through 0xFEFF.

The Windows operating system defines messages above the WM_USER range. Refer to your Windows documentation to ensure that you are not conflicting with these messages. OS/2 developers should also verify their user messages if developing portable applications.

The following figure shows the relationships between window, event, and handler classes.



1. Handlers are registered with the window.

2. Messages are encapsulated in event objects, which are passed to the window or control that had the event.
3. The window then invokes the handlers attached to it, passing the event object as a parameter.
4. The handlers are called sequentially with the most recently added handler invoked first. A handler indicates when processing for the event is complete by returning a Boolean value of *true*.
5. If none of the handlers process the event, it is passed up the owner chain.

The distinction between window classes and handler classes lets you separate the event-handling logic from the rest of the application. This enables reuse of this logic. For example, you can reuse a handler to verify telephone numbers wherever an entry field accepts telephone numbers.

Processing Events Using Handlers

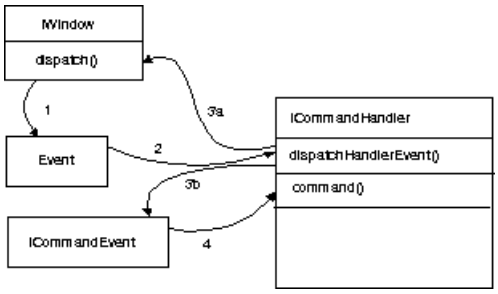
Each handler class has one or more virtual functions that are called to process the event. When an application processes events, it normally creates a class derived from a handler class, and overrides the virtual function to provide its own application-specific logic.

When you are within a handler member function, do not delete the IWindow object to which the handler is attached.

The order in which you attach handlers can cause a handler to not receive events because handlers are called in the reverse order that they are attached.

As a rule, ensure that handlers return from virtual functions within 1/10 second to avoid locking up the system by delaying the PM message processing.

The following figure shows how the ICommandHandler works. All handler classes contain a dispatchHandlerEvent function to determine whether the handler needs to process the event or return it. If the event needs processing, it creates the appropriate event object and calls the appropriate virtual function to process the event.



The numbers represent the following:

Number	Description
1	Window creates an event.
2	IEvent is passed to ICommandHandler.
3a	IEvent is not processed.
3b	ICommandEvent is generated.

Number	Description
4	ICommandEvent is processed by command().

The IEvent class acts as the base class for more specialized event classes. It provides general member functions to extract the message ID and message parameters. The IEvent derived classes generally add more specialized functions for extracting information specific to that type of event. The IEvent class provides a member function, setResult, for those events that require a value to be returned.

Chapter 14. User Interface Controls

Controls

A *control* is a part of the user interface that lets a user interact with data.

Controls are usually identified by text; for example, headings, labels in push buttons, field prompts, and titles in windows.

Static Text

Static text controls are text fields, bitmaps, icons, and boxes that you can use to label or box other controls. Your user typically does not interact with these controls using the keyboard or mouse. Generally, you do not need to change a static control's appearance on the screen, so visually these controls tend to be unchanging. Static text control classes include IBitmapControl, IIconControl, IOutlineBox, IGroupBox, and IStaticText. The IStaticText class creates and manages the static text control window.

You can set the text and its color, size, and position in the static text window.

Entry Fields

An *entry field* is a control window that enables a user to view and edit a single line of text. An entry field provides the text-editing capabilities of a simple text editor and is useful whenever an application requires a short line of text from the user.

If the application requires more sophisticated text-editing capabilities and multiple lines of text from the user, the application can use a Multiline edit field.

Applications typically use entry fields in dialog windows, although they can be used in non-dialog windows as well.

Multiline Edit (MLE) Fields

A *multiline edit* (MLE) field enables users to view and edit multiple lines of text. Use the IMultiLineEdit class to create an MLE field. The member functions of the IMultiLineEdit class enable you to display text files with horizontal and vertical scrolling, read a file into and save it from an MLE, or perform basic clipboard tasks (such as cut, copy, paste, and clear).

Button Types

A *button* is a type of control window used to initiate an operation or to set the attributes of an operation. A button can appear alone or with a group of other buttons. When buttons are grouped, you can move from button to button within the group by pressing the **Arrow** keys. You can also move among groups by pressing the **Tab** key.

A user can select a button by clicking it with the mouse or by pressing the **spacebar** when the button has the keyboard focus. In most cases, a button changes its appearance when selected.

© Copyright IBM Corp. 1998

A button is always owned by another window, usually a dialog window or an application's client window. It posts messages or sends notification messages to its owner when a user selects the button. The owner window receives messages from a button and can send messages to the button to alter its position, appearance, and enabled or disabled state.

To use a button in a dialog window, your application specifies the control in a dialog template in the application's resource-definition file. The application processes button messages in the dialog-window procedure. This is not supported on Motif.

There are four (three for Motif) main types of buttons, which determine how the button looks and behaves:

- Push buttons
- Radio buttons
- Check boxes
- Three-state check boxes

A radio button, check box, or three-state check box *control* an operation; a push button *initiates* an operation. For example, you might set printing options (such as paper size, print quality, and printer type) in a print-command dialog window containing an array of radio buttons and check boxes. After setting the options, you select a push button to notify an application that printing should begin (or be canceled). Then the application queries the state of each check box and radio button to determine the printing parameters.

A *push button* is a rectangular window that contains a text string. Typically, an application uses a push button to let the user start or stop an operation. A push button represents an action that is initiated when a user selects it. You can label it with text, graphics, or both. When a user selects a push button, the action occurs immediately if there is a handler for the generated command event.

Use the IPushButton class to create and maintain the push button window. By default, a push button generates an application ICommandEvent. You can change the default style by changing the window style value to generate a help event or a system command event. Using system command events is not recommended for portable applications. You cannot set colors for push buttons on the WIndows operating system

The command events generated by pressing the **Left**, **Center**, and **Right** push buttons are handled by the AHelloWindow::command member function. Note, the command events are the same as those used for the corresponding menu items. Therefore, the command function processing is the same whether you press the push button or select the item on the menu bar.

A *radio button* is a window with text displayed to the right of a small circular indicator. Use radio buttons to display a set of choices from which the user can select one. Each time the user selects a radio button, that button's state toggles between *selected* and *unselected*. This state remains until the next time the user selects the button. An application typically uses radio buttons in groups.

A group of radio buttons contains at least two radio buttons. Within a group, usually one button is selected by default. The user can move the selection to another button by using the cursor keys; however, only one button can be selected at a time. Radio buttons are appropriate if an *exclusive* choice is required from a fixed

list of options. For example, applications often use radio buttons to let users select the screen foreground and background colors.

The IRadioButton class lets you create and manage the radio button window. The ISelectHandler class processes the selection of a radio button. You add the handler to either the radio button or its owner window by calling the handler's handleEventsFor function.

Check boxes are similar to radio buttons, except that they can offer *multiple-choice* selection, as well as individual choice.

OS/2 When a user selects the choice, a check mark symbol appears in the check box to indicate that the choice is selected.

AIX When a user selects the choice, the check box is filled in to indicate that the choice is selected.

By selecting the choice again, the user deselects the check box. Use a check box to set a choice in a group of choices that are not mutually exclusive.

Check boxes also toggle application features *on* or *off*. For example, a word processing application might use a check box to let the user turn word wrapping on or off.

The ICheckBox class lets you create and maintain a check box. The selection of a check box is processed by using the ISelectHandler class. You add the handler to either the check box or its owner window.

Three-state check boxes are similar to check boxes, except that they can be displayed in *halftone* as well as selected and unselected. An application might use the halftone state to indicate that, currently, the check box is not selectable. This control is not supported on Motif. The selection of a three-state check box is processed by using the ISelectHandler class and adding the handler to either the three-state check box or its owner window.

If a three-state check box is selected, it can be either checked or halftoned. You have to use a combination of the isSelected and isHalftone member functions to determine the state of a three-state check box. The following table shows the state of the three-state check box and its relation to the isSelected and isHalftone member functions.

Current State	isSelected	isHalftone
checked	true	false
halftone	true	true
not checked	false	false

List Controls

A *list box* is a control that displays several items at a time, one or more of which can be selected by the user.

An application uses a list box when it requires a list of selectable fields that is too large for the display area or a list of choices that can change dynamically. Each list item contains a text string and an optional handle. The text string is displayed in the

list box window, but the handle is available to the application to reference other data associated with each of the items in the list.

The IBaseListBox class creates and manages list box control windows. Two types of list box controls are derived from IBaseListBox. They are IListBox and ICollectionViewListBox. IListBox extends the IBaseListBox list box control creation and management to include adding, removing, and replacing list box items. The ICollectionViewListBox<Element, Collection> template class extends the IBaseListBox control to enable the viewing of an ordered collection as items in a list box. The sequence of elements is the same between the ordered collection and the list box.

Combination Box Controls

A *combination box* is two controls in one: an entry field and a list box. There are three types of combination box controls:

- Simple
- Drop-Down
- Drop-Down List

This section describes combination box controls, also called *combination boxes* and *prompted entry fields*, which let the user choose and edit items from a list in an application.

Combination box controls enable the user to enter data by typing in the entry field or by choosing an item in the list box, unless it is a drop-down list, in which case, you cannot edit the entry field. Combination box controls manipulate the list box using the following member functions:

- IComboBox::add
- IComboBox::remove
- IComboBox::replaceAt

A combination box control automatically manages the interaction between the entry field and the list box. For example, when the user chooses an item in the list box, the combination box control displays the text for that item in the entry field. Then, the user can edit the text without affecting the item in the list box. When the user types letters in the entry field, the combination box control scrolls the list box contents so that items with those letters become visible.

Objects of the IBaseComboBox class create and manage combination box control windows. There are two types of combination boxes derived from IBaseComboBox. They are IComboBox and ICollectionViewComboBox.

Population of the combination box list box items is done by the derived classes. IComboBox contains the add, remove, and replace functionality, while ICollectionViewComboBox populates the combination box list box from collection elements using the setItems member function.

Slider Controls

A *slider* is a visual component that enables a user to set, display, or modify a value by moving the slider arm along the slider shaft.

A slider consists of a slider arm, one or two slider scales and, optionally, detents, tick marks, tick text, and slider buttons.

WIN Note that you can have two native slider scales, but only the primary one is visible.

The following table lists the slider components and their descriptions:

Slider Component	Description
Detent (PM and CCL control only)	A user-selectable mark that can be placed anywhere along the slider scale.
Progress indicator	A read-only version of a slider.
Slider arm	An arm that shows the current value by its position on the slider shaft and can be changed programmatically, as well as by users. Users can move the arm along the shaft to set slider values.
Slider buttons	Buttons that move the slider arm incrementally in the indicated direction.
Slider shaft	A track for the slider arm to move along.
Tick text	A label indicating the value the tick mark represents.
Tick mark	An incremental value in a slider scale.

Typically, sliders let users set values that have familiar increments, such as feet, degrees, or decibels. You can use sliders for other purposes when immediate feedback is required, such as to blend colors or show a task's percentage of completion. For example, your application might let a user mix and match color shades by moving a slider arm, or a progress indicator (with the ribbonStrip style) could show how much of a task is complete by filling in the slider shaft as the task progresses.

The slider's appearance and user's interaction with a slider is similar to that of a scroll bar. However, these two controls are not interchangeable because each has a unique purpose. A scroll bar scrolls information into view that is outside a window's work area, while the slider sets, displays, or modifies that information.

You can customize a slider to meet varying application requirements, while providing a user interface component that can be used easily to develop applications that conform to the Common User Access (CUA) user interface guidelines. Your application can specify different scales, sizes, and orientations for its sliders, but the underlying function of the control remains the same.

The ISlider class inherits from the IProgressIndicator class, which is a read-only version of the slider control. Typically, you use a progress indicator to display the percentage of a task that is complete by filling in its shaft as the task progresses. The default for progress indicators is to use the ribbonStrip style to fill the shaft. If you do not use this, there is a slider arm present to indicate the current value. Users cannot move the slider arm in a progress indicator.

Typically, you use the ISliderArmHandler class to handle events resulting from a user changing a control's input value without releasing the mouse, such as rotating the circular slider or moving the arm of a slider. ISliderArmHandler objects process input tracking events for the ISlider and ICircularSlider controls.

You create a handler derived from ISliderArmHandler and attach it to either the control whose input users can change or to the control's owner window. Call IHandler::handleEventsFor to pass the appropriate control window or owner window to the edit handler.

Spin Buttons

A *spin button* control is a visual component that gives users quick access to a finite set of data by letting them select from a scrollable ring of choices. Because the user can see only one item at a time, a spin button should be used only with data that is intuitively related, such as the months of the year or an alphabetic list of cities or states.

A spin button consists of at least one spin field and up and down arrows that are stacked on top of one another. These arrows are positioned to the right of the spin field.

Two types of spin buttons are derived from IBaseSpinButton. They are INumericSpinButton and ITextSpinButton.

You can create multi-field spin buttons for those applications in which users must select more than one value. For example, in setting a date, the spin button control can provide individual fields for setting the month, day, and year. The first spin field in the spin button could contain a list of months; the second, a list of numbers; and the third, a list of years.

The application uses a multi-field spin button by creating one master component that contains a spin field and the spin arrows, and servant components that contain only spin fields. The spin buttons are created during the construction of the master. When a servant spin field has the focus, it is spun by the arrows in the master component or by the cursor keys.

The value in a spin button entry field can be an element in an array of data or within a range of integers, defined by an upper and lower limit. Spin buttons that use arrays of data are text spin buttons and those that use a range of integers are numeric spin buttons.

Attach a handler derived from ISpinHandler to the spin button to capture spin events, such as the user pressing the up or down arrow.

Text-Change Events

WIN AIX Objects of the ITextChangeHandler class process text change events for the following controls:

- IComboBox
- IEntryField
- IMultiLineEdit
- IBaseSpinButton

Objects of the ITextChangeEvent class represent a pending change to the text data in an editable text control.

Text-change events describe a proposed change to the data in an editable text control. Such events result from a user attempting to change an input text value. It is not supported for OS/2 Presentation Manager.

Canvas Controls

A *canvas* is a window that manages its child windows. Various canvases provide a range of support, including the following:

- Managing the size and position of child windows
- Providing movable split bars between windows
- Supporting the ability to scroll a window

With the canvas classes, you can build windows with multiple child controls that contain fixed-size areas, user-sizeable areas, and scrollable areas. In addition, a canvas control lets you control tabbing between child controls, providing an alternative to using dialog boxes.

Generally, you build a complex window with a canvas control as the client area. This canvas can contain other canvas controls to build the desired layout.

The canvas classes are:

- ICanvas
  - IDrawingCanvas
  - IMultiCellCanvas
  - ISetCanvas
    - IToolBar
  - ISplitCanvas
  - IViewPort

The set and multicell canvases automatically size and position their child windows for you, based on the child window's minimum size.

Split Canvases

A *split canvas* places its child controls into panes. The panes are separated by movable or fixed split bars. (The default is movable split bars.) A split canvas can have its split bars oriented vertically or horizontally.

Note: In Motif, you can only move the split canvas using the small square buttons. However, in Windows and OS/2, you can move a split canvas using any part of the split bar line.

Use a split canvas to contain controls that can be resized to display more information, such as list boxes, containers, MLEs, and notebooks.

Note: Use the IListBox::noAdjustPosition style on a list box control in a split canvas because, otherwise, the operating system may adjust the height of the list box so it will not fill the height of the split canvas.

The order in which you create the child controls determines both their relative position on the split canvas and the order in which tab and cursor keys switch focus between them. For a canvas with vertical split bars, the child controls are arranged

with the control that was created first in the leftmost pane. For a canvas with horizontal split bars, the control that was created first is placed in the top pane.

Set Canvases

A *set canvas* arranges its child controls in either rows or columns. The Open Class Class Library uses the term *deck* for either a row or column. You can arrange the decks of a set canvas either horizontally or vertically. The set canvas attempts to place the same number of controls in each deck.

Each deck is created large enough to contain the largest control in the deck. To do this, the canvas calls the minimumSize member function for each child control. For controls that have sizes defined by the text they contain, such as push buttons and radio buttons, this default processing is normally sufficient. However, for a control that does not have a well-defined size, such as a list box or multiline edit control, you need to set its minimum size by overriding the calcMinimizeSize member function or by calling its setMinimumSize member function before adding it to the set canvas.

Note: Your application can determine the best minimum size for list boxes, multiline edit controls, containers, and frame windows.

The order in which you create the child controls determines their positions on the set canvas and the order in which tab and cursor keys switch focus between the controls. Several styles are available to control the orientation of the decks and the placement of controls within the decks. You can also alter the spacing between controls and between the decks and the edge of the canvas.

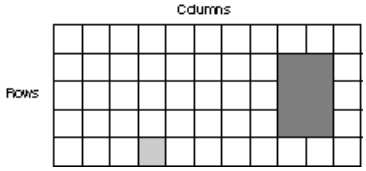
Multicell Canvas

A *multicell canvas* consists of a grid of rows and columns and looks like a spreadsheet. You place child controls on the canvas by specifying the starting cell and the number of contiguous rows and columns that they can span. You can refer to cells in the grid by the column and row value. The top left cell coordinate is (1,1).

The default cell size is 10 pixels high by 10 pixels wide. The actual number of rows and columns in the canvas is the highest row and column value used. For example, a radio button is placed at (4,5) and a push button at (2,7). Therefore, the canvas has 4 columns and 7 rows. Columns and rows can also be referenced by columnWidth and rowHeight.

The initial size of a row or column is determined by the size of the largest control in that row or column. By default, the row and column sizes are fixed. You can make the rows and columns expandable by using setColumnWidth and setRowHeight (setting the parameter expandable=true), in which case sizing the canvas also sizes them.

The following figure shows a multicell canvas. Notice that the small window's location is at column 4, row 5 (4,5), while the large window starts at column 9, row 2 (9,2). The large window is 2 columns wide and 3 rows long.



Note: A child window in a fixed-sized row and column is automatically sized based on its minimum size, which is typically based on the window's content and the available area within the window.

You can also leave rows and columns empty to provide spacing between child controls. If you do not explicitly size the empty rows and columns using setRowHeight and setRowWidth, they are sized to the default cell size.

You cannot have more than one window occupy the same 'starting cell'. (Starting cell means the start row and column you specify when you call addToCell.) However, you can have overlapping cells.

The size of a row or column is initially established by calculating the minimum size needed to hold all the windows in the row or column. In practice, this means the width of a column is set to the minimum width of the largest window in the column. The width is determined by calling IWindow::minimumSize on all windows in the column. The application can set the width or height of a column or row, respectively. If an attempt is made to set a value less than the minimum size of the largest window, the value is saved but ignored because the minimum size is the larger of the hard-coded size for the cell and the minimum size of the window. In other words, the application can reserve more space than the window needs but cannot cause the clipping of the window to occur.

A column or row never has a smaller width or height than the minimum size of the largest window in the column or row. If the windows later calculate a smaller minimum size, the width or height of the column or row is the larger of the following:

- The new minimum size of the largest window in the column or row
- The column width or row height set by the application

The minimum size of the canvas is based on the minimum sizes of all the windows on the canvas plus the size of empty rows and columns. If the area allotted to the canvas is less than this size, the canvas is clipped at the lower right corner. You can use the IViewPort class to add scroll bars to handle this situation.

The multicell canvas has two styles that can help you create layouts with a multicell canvas by providing a way to easily visualize the placement and size of each child window. The gridLines style causes the multicell canvas to have grid lines between the rows and columns of the canvas. Grid lines are stationary. If you want grid lines that you can move with the mouse, use the dragLines style, which causes the multicell canvas to have draggable grid lines between the rows and columns of the canvas.

Here are some considerations when using grid lines and drag lines:

- Grid lines and drag lines are placed at the left and top edges of cells.
- Child windows can overwrite grid lines and drag lines.
- The use of grid lines or drag lines does not change the initial placement or sizing of child windows.
- Moving a drag line causes the child windows on each side of the drag line to be resized in a manner similar to that of an ISplitCanvas.

View Ports

A *view port* canvas provides a scrollable view area with horizontal and vertical scroll bars. By default, the scroll bars display only when needed. A view port can have

only one child control. The size of the child control is fixed. If the view port is smaller than the child control, the view port allows the user to scroll the child control.

If you need more than one control in a view port, place the controls into another type of canvas, which you can then make the child of the view port.

Drawing Canvas

The IDrawingCanvas class is a control class that provides support for drawing graphic objects. The drawing canvas contains an IGraphicGroup to which you can add graphic objects. By default, a paint handler is attached to the drawing canvas. When paint events occur, the paint handler sets the clip region to the drawing canvas' update region and iterates through the IGraphicGroup, redrawing the graphic objects necessary to update the window. If you do not add graphic objects to the IGraphicGroup, the drawing canvas simply paints the window using the current background color.

If you want to use your own paint handler instead of the drawing canvas' default paint handler, construct the IDrawingCanvas object without the useDefaultPaintHandler style. The default paint handler is not attached to the drawing canvas.

You have the option of setting the graphic context that the drawing canvas will use when drawing the graphic objects contained in the IGraphicGroup object. By changing graphic contexts, you can change the appearance of the graphic objects or the device on which they are rendered.

To add graphic objects to a drawing canvas, create an IGraphicGroup object and call setGraphicList. The drawing canvas iterates through the drawing canvas for its IGraphicGroup. You can easily change the objects drawn in the drawing canvas by changing the IGraphicGroup object.

If the drawing canvas is resized smaller, it does not repaint. If your application requires the drawing canvas to repaint in this case call the refresh member function.

Notebooks

A *notebook* control is a visual component that organizes related information on individual 'pages' so that a user can find and display that information. It simulates a real notebook and provides the user with a recognizable visual component.

WIN The native Windows tab control consists of a folder with tabs.
OS/2 The CUA '91 notebook consists of a notebook client area, a binding, page buttons, optional tabs, and an optional status area.

Users select and display pages using a mouse or the keyboard. You can customize this notebook to meet varying application requirements, while providing a user-interface component that can be used easily to develop products that conform to the Common User Access (CUA) user-interface guidelines. Your application can specify different colors, sizes, and orientations for its notebooks, but the underlying function of the control remains the same.

The INotebook class supports both the CUA '91 notebook control and the Windows tab control. The CUA notebook control is a ported version of the OS/2 Presentation

Manager notebook control. It has the same look-and-feel as the OS/2 Presentation Manager notebook control. The Windows tab control is a control that is native to the Windows environment.

Tab Controls

WIN OS/2 The ITabControl class gives you another option to implement a settings or properties dialog. ITabControl creates a window that appears as a stack of tabbed pages with an optional button area.

For OS/2, this control uses the BKS_TABBEDDIALOG style on the PM notebook. On the Windows operating system, this is the native tab control with some additional drawing to emulate the Windows property sheet.

You can use an ITabControl object as the client of an IFrameWindow to create a tab control dialog.

An ITabControl object manages a set of ITabPage objects and an optional set of IPushButton objects. You must create an ITabPage object or an object of a derived class for each page in the tab control. A tab page is a logical representation of a page in a tab control and has an associated page window and optional push buttons. A tab page object is not a wrapper for a platform window as are IWindow derived objects.

Each tab page has a style and either text or a bitmap to be displayed on the corresponding tab in the tab control. You can specify the text and bitmap for the tab page during ITabPage construction or by using one of the ITabPage Tab Text & Bitmap member functions. In the Windows environment, tab pages can have both text and a bitmap displayed on the tab simultaneously. In the OS/2 environment, you can assign both text and a bitmap for a tab, but only the last item set is displayed.

The optional push buttons in the tab control appear in a button area below the tab pages, or, in the OS/2 environment, the button area can optionally appear above the tab pages. You can specify buttons on a per page basis as well as specifying a set of common buttons. ITabControl displays the set of common buttons whenever the top page has no page specific buttons. You must manage common buttons via ITabControl member functions and manage page specific buttons via ITabPage member functions. To process the command events generated by tab control buttons, create a handler of an ICommandHandler derived class and attach it to either the tab control or frame window.

You can add one of the five predefined push buttons provided by the EButton enumeration as common or page specific buttons. You can use the same buttons in multiple page button sequences as well as the common button sequence. This allows maximum flexibility in tailoring the buttons that appear as the user selects each tab page. For example, one tab page may have Ok, Cancel, and Help buttons while all other pages have Ok, Default, Cancel, and Help buttons. You would add the Ok, Default, Cancel, and Help buttons as common push buttons and add the Ok, Cancel, and Help buttons as page push buttons for the unique page.

You can also create your own IPushButton objects for use as tab control buttons. You must use the window returned by buttonParent as the parent of any IPushButton objects you construct for use as either common buttons or page buttons.

To process the command events generated by tab control buttons, you must attach a handler derived from ICommandHandler to the tab control or frame window.

Containers

A *container* control holds objects. For example, OS/2 provides a variety of containers, such as folders, templates, and the Workplace Shell itself. Containers can display their objects in different views: tree, icon, text, name, and details views. Using the Open Class Class Library, you can also develop your own containers and change their views, behaviors, and layouts.

 The IContainer class supports both the CUA '91 container control and the Windows container. The Windows CUA container is a ported version of the OS/2 Presentation Manager container control. It has the same look-and-feel as the OS/2 Presentation Manager container control. The Windows container is composed of the list view and tree view controls which are native to the Windows environment.

Use the style, IContainerControl::pmCompatible, to select the CUA '91 container control. The native Windows list view control is the default selection in the Windows environment.

AIX Due to the limited capabilities in Motif, the following areas of the IContainerControl and its related classes are not supported for the Open Class Class Library for AIX release:

- Vertical split bar in the details view
- All emphasis, except selection and cursor emphasis
- The following ICnrHandler overrides will not be called:
  - deltaReached
  - windowScrolled
- Draw item (for example, owner draw)
- Scrolling functions, including:
  - scroll
  - scrollDetailsHorizontally
  - scrollHorizontally
  - scrollToObject
  - scrollVertically
- IContainerControl constructors for creating an instance from an existing container control or a dialog template

Clipboard

A *clipboard* is a system-wide place for users to store data temporarily. The clipboard enables your user to move data within a single application or to exchange data among applications. Typically, a user selects data in the application using the mouse or keyboard, then initiates a cut or copy operation on the selected data. The clipboard can hold an entire object or part of that object, and it can hold any kind of object. For example, the clipboard can hold a single line of text or an entire database, a single line segment or an entire graphic.

When the user selects the paste operation, the data is transferred to the application from the clipboard.

Note: Only a single item of data can be stored in the clipboard at a time. Therefore, do not use the clipboard to store data unless a user requests it because you can overlay the user's data stored there. This is important: the user must always control access to the clipboard.

While you can only store a single item of data in the clipboard, you can store this item in multiple formats. This allows an application to choose the format it supports that gives it the most information about the data. For example, a graphics application might copy a picture into the clipboard as both a metafile and a bitmap. This allows applications that support both metafiles and bitmaps to retrieve the picture as a metafile if it needs to modify the picture or as a bitmap if it only needs to display the picture.

IClipboard predefines several system clipboard formats. In addition, any application can create and register additional private formats.

Before you can write any data to, or read any data from, the clipboard, you must first open it. Only a single application at a time can open the clipboard. If an application tries to open the clipboard but another application already has it open, it waits until the clipboard is available. The default behavior of the clipboard classes minimizes the time the clipboard is open.

If you use the default behavior of IClipboard, the clipboard functions that require an open clipboard open it when needed and close it when finished. You turn off the default behavior of IClipboard when you explicitly open the clipboard by calling IClipboard::open. If you open the clipboard in this manner, functions in IClipboard will not close the clipboard when complete. If you explicitly open the clipboard, you must close the clipboard by calling IClipboard::close. You can turn off the default behavior of IClipboard to place different formats of your data on the clipboard without opening and closing it to write each format.

All clipboard operations must be associated with a window. You provide this window on the IClipboard constructor. If necessary, IClipboard makes this window the *owner* of the clipboard. The clipboard owner is the window responsible for the data put on the clipboard. It is also the window that the operating system sends messages to for events relating to the clipboard. The IClipboard object establishes this window as the system clipboard owner when you call IClipboard::empty. If you call IClipboard::owner before calling empty, your window will not be returned because it is not yet the system clipboard owner.

The clipboard classes support an advanced concept called *delayed rendering*. Delayed rendering allows you to wait until another application requests the data before you put the data on the clipboard. You activate delayed rendering by supplying 0 for the data when you call the clipboard functions to place data on the clipboard.

You process clipboard events by creating and attaching an IClipboardHandler object to your clipboard owner window. In particular, if you use delayed rendering, you must attach an IClipboardHandler object to your clipboard's window (the owner window). The window dispatcher calls this handler when a request is made to the clipboard for data that has not been placed there yet.

Because the clipboard should only be kept open for a short time, create IClipboard objects as temporary objects with a short lifetime. This helps ensure that the clipboard is only open for the time necessary.

The IClipboard destructor always closes the clipboard if it is still open.

Toolbars

A *toolbar* is a window whose buttons represent tools or menu items and actions. The toolbar can be located along the top, bottom, or sides of a frame window or can 'float' and be positioned anywhere on your desktop. The floating toolbar can then be moved independently or you can 'pin' the toolbar to the frame window. You can also have multiple tool bars with a variety of different toolbar buttons using text, bitmaps, or both.

A toolbar button class inherits from ICustomButton and provides drawing behaviour for the buttons in a toolbar. These buttons accept text and a bitmap and can draw bitmaps transparently without the need for a mask bitmap. Bitmaps used in these buttons must be created with one color reserved as the transparent color. By default this color is pink (255, 0, 255) but it can be changed on a per button basis.

The classes that comprise the toolbar are also shipped with common text and bitmaps for frequently used functions found on a toolbar. This ensures a common look among applications and products. In addition, toolbar buttons support a standard style to ensure a consistent look across applications.

The toolbar is essentially a frame extension with buttons that represent common actions. You can add both fly-over help and drag-and-drop support on the toolbar. You can use the direct manipulation support to rearrange the toolbar buttons within an application or add new buttons from a menu (PM only).

The IToolBar class creates and manages a toolbar area for a frame window. You can use the following classes to create tool bars:

IToolBar
Creates a toolbar, positions that toolbar as a frame extension or as a floating toolbar, and adds and removes toolbar buttons.

When you construct controls for a toolbar, you must explicitly add them to the toolbar using one of the following member functions:

- IToolBar::addAsFirst
- IToolBar::addAsLast
- IToolBar::addAsPrevious
- IToolBar::addAsNext

IToolBarButton
Creates and manages the toolbar button window. IToolBarButton provides support for displaying a bitmap, text, or bitmap and text on a toolbar button. If you specify a particular style for the view of your toolbar button, this style is changed when the button is added to the toolbar if it differs from the toolbar style itself.

The following classes are used by IToolBar and IToolBarButton to implement a toolbar:

IToolBarContainer
Provides a set canvas for layout control for multiple toolbars within a frame extension. Objects of this class are created automatically by IToolBar objects to house one or more toolbars.

IToolBarTitle

Creates title bars that are smaller in height than the system title bar. This is created automatically whenever you create a floating toolbar using the IToolBar class.

IToolBarFrameWindow

Represents a frame window to contain floating toolbars. IToolBarFrameWindows are created when you construct a floating toolbar.

WIN The maximize and restore button on an IToolBarFrameWindow allows you to expand and collapse the toolbar contained in the toolbar frame window. You can shrink the toolbar to recover screen area when you are not actively using the toolbar and later expand the toolbar to show all toolbar controls.

OS/2 IToolBarFrameWindows are visually distinct from other PM frame windows. The title bar height of IToolBarFrameWindows are smaller than other frame windows. In addition, this small frame window contains two buttons that appear where the frame window's minimize and maximize buttons normally are.

The left button is a toggle button that allows you to pin or attach a toolbar frame window to its owning frame window. This keeps the toolbar in the same location relative to its owner frame window when the owner is moved.

The right button allows you to expand and collapse the toolbar contained in the toolbar frame window. You can then reduce the size of the toolbar when it is not in use. Select the button again to expand the toolbar and show all the toolbar controls.

ICustomButton

Provides the base class for IToolBarButtons and creates and manages the custom button window. Custom buttons allow the application to customize the button appearance by providing an ICustomButtonHandler to draw the button.

ICustomButtonDrawEvent

Provides information about a custom button event for ICustomButtonHandler event-handling functions.

ICustomButtonDrawHandler

Processess ICustomButton drawing events. Attach objects of this class only to ICustomButton objects.

Chapter 15. Application Resources

Resources

You define resources in a resource file. A *resource file* is a file that contains data that your application uses, such as text strings and icons. For example, you can define a menu, string table, or dialog template in the resource file. You also define the string ID, corresponding to each static string you use in a window, in the resource file.

The resource compiler produces a compiled version of the resources, which is then incorporated into the application's executable code or stored in a dynamic link library (DLL) for use by one or more applications.

A benefit of defining resources in a resource file is that you can make changes to resource definitions without affecting the application code itself.

You can also provide national language versions by storing the resources for each language in a separate resource file. You can then build your application as separate executable versions for each language (each with a different resource file bound to it) or as a single executable with a separate DLL for each language.

Window Resources

Window resources are read-only data segments stored in an application's .exe file or in the dynamic link library's .dll file.

Predefined Open Class Class Library window resources include keyboard accelerator tables, icons, menus, and bitmaps.

Most window resources are stored in a format that is unique to each resource type. The system translates the formats, as necessary, for use in functions.

To access window resources, you must prepare a resource file (ASCII file with the extension .rc). Then the ASCII resource file must be compiled into a binary image using the resource compiler. The compiled resource file (binary file with the extension .res) can be linked to your application's .exe file or to a dynamic link library's .dll file.

Dialog Templates

A *dialog template* is a data structure that describes a dialog window and its control windows. Windows and PM uses the data in the dialog template to create the dialog window and control windows. An application can create a dialog template at run time, or it can use the system resource compiler to create a dialog-template resource.

Motif does not support dialog templates. If you write portable applications, use canvases instead of dialog templates. The Hello World 4 sample application shows you how to do this.

© Copyright IBM Corp. 1998

NLSPATH

IDynamicLinkLibrary and ICurrentApplication can be constructed to load a resource library from the NLSPATH, so that on a single system you can have multiple language versions of your resource libraries. The constructors load the correct library based upon the LANG and NLSPATH values.

Dynamic Data Exchange

WIN OS/2 *Dynamic data exchange* (DDE) is a client/server protocol for communicating between two applications running on the same machine. *Client/server* implies a relationship between two applications where the *client* sends requests to the *server*. The server handles the requests and provide services to the client application which then consumes those services. The server-provided services can be either of the following:

- Data
- The ability to execute commands on behalf of a client

The server can provide these services to one or more clients, and a client can request services from one or more servers. In addition, an application can be a client and a server because it can request services from a server while providing services to clients.

As a client/server protocol, DDE enables data to be dynamically exchanged, and thus shared, between two applications running on the same machine. Applications are shielded from the operating details and can share data once they agree on the type of data being exchanged. DDE works with applications written for Microsoft Windows and OS/2 Presentation Manager, or a combination of both.

DDE is based on a client/server application model, as follows:

1. The DDE client application requests a conversation with the DDE server application.
2. The DDE server accepts and handles the conversation.
3. The DDE client can now send a variety of predefined requests to theserver.
4. The DDE server then sends data, as well as positive and negative acknowledgments, to client applications.
5. The only unsolicited communication a DDE server application can have with a client application is to end a conversation. It cannot send requests to the client application.

From the users' perspective, DDE gives them the ability to import and export data between applications. DDE applications can exchange data on an ongoing basis without the users intervening. From the application developers' perspective, DDE is a client/server-based protocol consisting of a set of messages, macros, and data structures.

Typically, in a multiprocessing operating system, messages deliver requests and replies. The following points are true:

- Resources in one application are protected from being accessed by another application.
- Memory protection schemes create obstacles when applications need to share data.

To counteract these points, the OS/2 operating system provides*interprocess communication* (IPC), which enables information to be exchanged between processes or threads through semaphores, queues, pipes, and shared memory. For these IPC methods to work, each application must do the following:

- Register the IPC method being used
- Contain specific information on how to access the data

In windowed environments, such as Presentation Manager or Windows, the primary IPC method is to pass messages. In this case, the same memory protection that protects the operating system also protects the data passed in these messages. Thus, for one application to send data to another, it allocates a *shared memory object* and gives it to the receiver.

How the DDE protocol works

The Open Class Class Library DDE protocol works with the operating system APIs because DDE messages are dispatched like other operating system messages. Because DDE uses object windows to communicate, an application that uses this component needs to process window messages. To do this, execute IThread::processMsgs, even if your application has no interface components. Call `IApplication::current().run()`.

Establishing DDE Conversations

When you use DDE, the client application must first initiate a conversation with the server applications. Conversations are based on a two-tier hierarchy: *application* and *topic*. Application is typically the name of the executable program; the topic is typically the name of a file in a spreadsheet or word-processor application. During the initiation, the server determines if it can satisfy the client's request. The server does not respond to the client if it cannot satisfy the request. If it can satisfy the request, it creates a new window and includes the handle of this new window in its response. A server responds once for each topic it can support and, for each response, the server provides a unique window handle. This guarantees that each window handle pair identifying a conversation link is unique.

Once a DDE conversation is established, client applications can request services from the server through *DDE transactions*. In DDE, all transactions are one-way because the client always issues the request and a server responds. The only exception to this rule is the end transaction. The client specifies what it wants from the server by providing the data item name of a service.

A conversation continues until one of the participants ends it. The conversations can be ended by either participant at any time during the interchange.

DDE Transactions

DDE is transaction-based because all communications between DDE client and server applications occur within the context of a DDE conversation. Once a DDE client application has successfully initiated a conversation with a DDE server application, all subsequent communications are in the form of asynchronous transactions. These transactions are asynchronous in nature, because they are implemented using operating system messages that are posted, not sent, to the other application. This can lead to complications because there is a minimal amount of information provided in the DDE messages and control blocks they pass in

shared memory to connect a response to a particular request. As a result of these potential complications in any asynchronous communications, the DDE protocol requires server applications to respond to requests from any one client in the exact order they are received. With the Open Class Class Library, the DDE framework enforces this synchronization for you on the server side. On the client side, the framework keeps track of all outstanding transactions, ensuring they are responded to in the correct order. It also provides you with information about your original request, along with the response data.

Using Applications, Topics, and Items

For the purpose of initiating a DDE conversation, the conversation is uniquely identified by the application name of the DDE server and the name of the *topic*, where a topic is a logical data context. For example, for a word processing or spreadsheet DDE server application, the topic is typically the file name of a document or spreadsheet. For a DDE news server, the topic could be an area of interest. Once a conversation has been initiated, it is uniquely identified by a pair of window handles, provided by the client and server applications. An item is a named data object, which can be passed as part of a transaction within a DDE conversation. An item can be anything from a simple data type, such as an integer or character string, to something as large as a bitmap or data file.

Specifying Data Formats

Whenever a DDE application requests or provides data, it must specify the format of the data. With this information, DDE applications know how to package, unpackage, or interpret the data they pass between them. Before a data format is used by a DDE application, the application must ensure the data format has a unique identification by registering its name in the system atom table. The operating system defines a number of constant names for industry-standard data formats, and adds them to the system atom table. The Open Class Class Library DDE framework verifies that any format specified by your application is defined in the atom table. As a DDE client application, you request data in a format supported by the DDE server application. As a DDE server application, you provide data in useful formats that DDE client applications can process.

The standard format for exchanging text data is the Developer's Toolkit for OS/2 constant, DDEFMT_TEXT, which is defined to be #1.

Understanding the DDE Framework Design

On the DDE server side, IDDETopicServer is the primary class, and represents a DDE server for a single topic. An IDDETopicServer object can have as many concurrent conversations as it needs. These conversations can be with the same or different applications. The only restriction is they must all be on the same topic.

On the DDE client side, IDDEClientConversation is the primary class. An IDDEClientConversation object represents a single conversation with a single DDE server application. An IDDEClientConversation object can be reused. For example, when you end a conversation with a DDE server application, you can use the IDDEClientConversation object to begin a conversation with any DDE server application on any topic.

Most IDDEClientConversation and IDDETopicServer functions have a default behavior, which you can override on each callback function. For an

IDDETopicServer object, the callback functions pass requests for information or for the server to carry out an action. The default implementation for most functions is to return *false*, indicating the DDE server does not support this particular DDE transaction type. To support a particular transaction type, you must override the function and provide an appropriate implementation. This typically involves setting information or data into the passed DDE event. The IDDETopicServer::requestData function is a pure virtual function so you must derive a class from IDDETopicServer to provide an implementation for it. In this way, you can build a DDE server application incrementally, supporting only the DDE transaction types you need for your application.

For an IDDEClientConversation object, the callback functions pass you the responses that the DDE server application has sent to your requests. The default implementation for these functions is to do nothing. Because you made the request to the DDE server application, you are interested in the response the server has sent to you. You must override these functions to process the data and information that is sent as part of each response. IDDETopicServer and IDDEClientConversation objects pass data and information to you in the form of event objects.

IDDEEvent is the main event class from which most other DDE event classes are derived. The only exceptions to this are the event classes associated with beginning and ending a conversation; IDDEBeginEvent, IDDEEndEvent, and IDDEClientEndEvent.

Chapter 16. User Interface - Direct Manipulation

Direct Manipulation

Direct manipulation is a user interface technique that lets a user start application functions by manipulating objects. The user begins an action by moving the mouse pointer over an object and then pressing and holding down the drag mouse button (mouse button 2 is the default for OS/2) while *dragging* the selected object to a new location. The user then *drops* the object onto the new location by releasing the mouse button. For this reason, direct manipulation is also known as *drag and drop*.

Thus, the user can perform operations directly on objects that appear on the desktop or within an application.

Direct manipulation is not limited to objects in containers, as the object can be a text string in an entry field. Also, users can drag and drop an object onto a new location in the current window, onto another object in a window, or onto a different window.

Direct manipulation is used to move, copy, and link objects. Generally, move is the default operation and is accomplished through the use of the mouse button defined for drag and drop. Other operations can be specified with the additonal use of augmentation keys: the Ctrl key or a combination of the Shift and Ctrl keys.

To initiate a move operation, the user presses and holds the Shift key while moving the drag mouse button. The visual indication of this operation for OS/2 and Motif is the drag image. For Windows, the visual indication for move is the halftoning of the object.

To initiate a copy, the user presses and holds down the Ctrl key while pressing and moving the drag mouse button. The visual indication of this operation for OS/2 and Motif is the halftoning of the drag image. For Windows, the visual indication for copy is the halftoning of the object with the plus sign displayed in the lower-right corner.

Likewise, users can accomplish a link operation by pressing and moving the drag mouse button while pressing the Shift and Ctrl keys. The visual indication of this operation in OS/2 is a line that is drawn that connects the drag image with the object where the drag was initiated. For Windows, the visual indication is the halftoning of the object with the short cut arrow displayed in the lower-right corner.

Users can request help during a direct manipulation operation by pressing the help key F1. This displays help for the object a user is dragging over.

To cancel a direct manipulation operation, press the Esc key.

The Open Class Class Library provides four main classifications of objects to support direct manipulation:

- A drag item (IDMItem)
- A drag item provider (IDMItemProvider)
- A renderer (IDMSourceRenderer or IDMTargetRenderer)
- An event handler (IDMSourceHandler or IDMTargetHandler)

© Copyright IBM Corp. 1998

The collaboration of these objects allows the rendering from a source to a target location. *Rendering* is the process by which data is transferred from the source of a direct manipulation operation to a target. The following is an overview of the process:

1. A user initiates a drag request, which generates an event that is processed by the source handler.

   For Windows, this event needs to be generated. Windows leaves it up to the control to decide when a drag operation is occurring. This decision is usually made on a per control basis because Windows allows using the same mouse button, the left button by default, for selection and drag. IWindow::isDragStarting (Windows only) helps to detect a drag operation and is called for controls that have been enabled for drag through IDMHandler::enableDragFrom or IDMHandler::enableDragDropFor. After a control has been enabled for drag, IWindow::isDragStarting is called for every mouse button down, up, or move that is dispatched to the control. When a drag has been detected, the isDragStarting override should return true. The return value of true causes IDMHandler to send a WM_BEGINDRAG message and initiate the IDM classes to start the drag operation.

   Once the WM_BEGINDRAG message is generated, the IDM classes work the same on both the Windows and OS/2 platforms. OS/2 can generate the WM_BEGINDRAG message for you, because a different mouse button is used for selection and drag operations. Windows also offers you the option of initiating a drag with the right mouse button for non default drag and drops, which allows differentiating the drag and selection operations. IWindow has a default implementation of isDragStarting that tracks the left button state along with mouse move messages. If the left button down message is received and the mouse moves more than 2 pels before a left button up event is received, then a drag operation has occurred. If you need to support drag from a control that requires more elaborate drag conditions, override this member function and supply an implementation. You can use IWindow, IEntryField, IMLE and IContainerControl implementations of isDragStarting for examples.

2. The source handler uses the source window's attached drag item provider to request generation of source drag items.

   Drag items represent the objects that are the focus of the direct manipulation operation and provide access to the object's data. *Drag item providers* are designed to assist in the generation of drag items and bridge the gap between the drag items and the source and target handlers.

   When you use the IDMItemProviderFor template to instantiate the provider, the static function in the derived drag item class, generateSourceItems, is called. Alternatively, if the provider is instantiated from a derived drag item provider class, then the IDMItemProvider::provideSourceItems override is called.

3. Once the source drag items are generated, appropriate source renderers are selected.

4. When you enter a potential target window, an event is generated that is processed by the target handler.

5. The target handler uses the target window's attached drag item provider to request generation of target drag items.

   When you use the IDMItemProviderFor template to instantiate the provider, the target drag item constructor is called, as follows:

   ```
   IDMItem ( const Handle& item );
   ```

   Alternatively, if the provider is instantiated from a derived drag item provider class, then the IDMItemProvider::provideTargetItemFor override is called.

6. The target handler also uses the drag item provider to request additional verification support via the virtual function, IDMItemProvider::provideEnterSupport.
7. Once the target drag items are generated and verified, the appropriate target renderer is selected.
8. The drop is processed based upon the rendering mechanism and format, which is stored in the target renderer, and the data is subsequently transferred. The virtual function, IDMItem::targetDrop, is used to process the drop event.

Renderers encapsulate the various mechanisms and formats that are used for rendering and implement the logic that supports the mechanisms and formats (RMFs). Generally, a *mechanism* defines the method of data transfer, whereas the *format* identifies the format of the data. The Open Class Class Library's implementation groups the mechanisms and formats under one mechanism.

Target rendering transfers data from the source to the target. Because the source can package all the required transfer information when the drag operation begins, the target can complete the drop operation without further assistance from the source.

A good example of target rendering involves the use of the Open Class Class Library's process RMF. If the source and target are in the same process, the target directly accesses the source's data and subsequently renders the information as required.

On the other hand, *source rendering* occurs when the target requires additional information from the source in order to complete processing of the drop. The target issues the appropriate events to request the information from the source. A good example of source rendering involves the use of the User Interface Class Library's shared memory RMF. When the source and the target are in separate processes, the target can select this RMF that generates a shared memory buffer that, in turn, transfers the data between the source and target.

Understanding Drag Items

Drag items are represented by objects of class IDMItem. *Drag items* encapsulate the logic that serves as the bridge between the context-insensitive handlers and renderers and the application-specific behavior of particular source and target windows. Thus, the drag items provide the application-specific semantics of the direct manipulation operation.

IDMItem is the base class that defines the general behavior of all direct manipulation items. The three derived classes provided by the Open Class Class Library provide specializations of the base class that represent the objects being dragged and dropped on specific controls.

The following classes are derived from IDMItem:

- IDMCnrItem
- IDMMLEItem
- IDMEFItem

Understanding Drag Item Provider

The IDMItemProvider class is an extension of the IWindow class that provides direct manipulation functions. Objects of the IDMItemProvider class allow generic controls, such as an entry field, to generate context-sensitive drag items. For example, a container that contains customer objects can generate a 'customer' item; a bitmap can provide an item that can extract the picture from a .BMP file.

The IDMItemProvider class also provides functions that deal with target entry and exit, as well as help. With IDMItemProvider::provideEnterSupport you can verify objects, using different schemes, when the object is over a potential target.

For example, if you want to restrict drops in an icon view container to the white space area of the container, you can use the provideEnterSupport function to determine if the pointing device is over either white space or an actual container object:

```
IBase::Boolean CnrProvider::provideEnterSupport ( IDMTargetEnterEvent &event )
{
  /*********************************************************************/
  /* Allow default verification to occur                               */
  /*********************************************************************/
  if (Inherited::provideEnterSupport( event ))
  {
    /*******************************************************************/
    /* Prevent the drop if we're over a container object (icon view)     */
    /*******************************************************************/
    if (event.object())
    {
      event.setDropIndicator( IDM::notOk );
      return(false);
    }
    return(true);
  }
  return(false);
}
```

Note: The return value is not currently used to determine if a drop is allowed. Only the drop indicator is used to set the drop disposition.

Chapter 17. Graphics

2D Graphics Overview

The 2D Graphics Framework is a self-contained collection of C++ classes within IBM Open Class which provide classes for creating, manipulating, and rendering 2-dimensional graphics. You can use 2D Graphics classes to build applications that create and manipulate 2-dimensional graphics and images. These classes are easy to use. You instantiate the classes you need and call their member functions. You do not need to know linear algebra or any other advanced mathematics to effectively use these classes. All advanced mathematics is implemented for you.

You can extend most of these classes to add your own functionality by deriving from the abstract base classes, implementing the pure virtual functions, and adding whatever behavior you need.

The 2D Graphics Framework classes can be used to perform the following:

- Creation of graphical user interface (GUI) elements and decorations
- Manipulation of pixel-base images, screen captures and so on
- Creation of complex 2D graphics composed of groups of simple graphics used for modeling

The 2D Graphics Framework is grouped into the following main categories:

- Geometry
- Graphics Attributes
- Transformation
- Rendering
- Modeling

The graphics and geometry classes provide functionality for different areas of graphics programming:

- Graphics classes are derived from IMGraphic. They can own a geometry class and can have functions to transform and otherwise manipulate the geometry.
- Geometry classes have no common parent class. They have functions that let you manipulate the geometry, but are not intended to be derived from.

You can use graphics and geometry classes together, or use the geometry classes by themselves. You can also extend the graphics classes by creating your own graphic class. You can design your own class to use as many geometry classes as you want.

The geometry classes provide many ways to construct a shape because you construct the geometry first and then use it to construct the graphic. The graphics classes constructors give you the option of including an attribute bundle to define characteristics of the graphic.

Transformation functions let you rotate, translate, scale, or apply a general transformation to the geometry without using matrix multiplication.

The graphic classes provide mechanisms to determine the bounds of a graphic as a rectangle enclosing the shape. This bounding box can be used, for example, to

© Copyright IBM Corp. 1998

determine whether a user has selected a graphic (hit detection). The supported functions return a bounding box that does not have a tight fit.

When you use a graphic class to create a shape, you get the following:

- The ability to display the graphic with user-defined attributes state class such as an IGrafBundle.
- The ability to combine several graphics into a group for common operations.

Geometry

The geometry classes serve as the foundation for creating graphics. Use these classes to describe pure geometric shapes.

The geometry classes are grouped into the following categories:

- Computational geometry: Computational geometry classes, including points and collections of points, are used for 2D geometric calculations and as building blocks for other 2D geometry classes.
- Path geometry: Path geometry classes describe shapes of open shapes that need not be filled.
- Area geometry: Area geometry classes let you perform operations on a set of basic 2D geometry to create an arbitrary 2D shape. Area geometry classes describe shapes of an enclosed path that can be filled.
  Some geometric shapes, such as point, line, and nonenclosing polyline and curves, are of path geometry. Enclosing geometric shapes such as rectangle, circle, ellipse, polygon, and enclosing curves (loops) can be both path and area geometry.
- Special area geometry: Special area geometry classes, used primarily for clipping, make area geometry by concatenating different geometric shapes. These classes let you perform constructive area geometric operations through boolean operations such as union, intersection, and subtraction.
- Image geometry: Image geometry classes let you handle and manipulate 2D images (bitmaps). These classes convert basic geometry from coordinate units to pixels. 2D attribute bundles have special properties for determining the color of an image (transfer modes) and for specifying filters to control the appearance of an image when it has been enlarged or shrunk.

NOTE: For immediate mode rendering, a user can create geometry and graphics attributes and render directly to the graphics rendering/display system, the rendering pipeline, without using the modeling classes. However, IMGraphic modeling classes are more convenient for retained mode rendering where a user needs to attach attributes and transformation functions to a geometry.

Attributes and Bundles

Graphics attributes classes describe primitive attributes such as paint, pen, and image transfer mode. These classes collect combinations of such primitive attributes to use when rendering the geometry.

Transformations

Transformation classes, specifically IGrafMatrix, define transformation matrices for manipulating graphics geometry and modeling.

Modeling

Modeling classes associate geometries with graphic attributes, and transformations for simple graphics and hierarchical collections of graphics. IMGraphic is the abstract class of IGraphicGroup and its related classes. Use these classes to create complex graphics models (scenes), and extend these classes to enable graphics metadata storage. The virtual draw function lets you render the geometry of a drawable graphic with attributes and transforms applied.

Rendering

The rendering pipeline, controlled through IGrafPort, performs imaging operations. You associate a graphic and its state information, including its attribute state, transformation matrix, and clipping state, with a grafport. The grafport renders the graphic from this state information.

IGrafPort is associated with IGrafDevice. For performance reasons, IGrafDevice is at the level of abstraction of platform/device-specific code. The concrete subclass of IGrafDevice is platform/device specific. IGrafDevice encapsulates the nonportable code: the device-dependent layer is completely hidden from the client.

Prerequisite Information

For background information on graphics programming, refer to a dictionary of graphics terms or to the following books on graphics programming:

- Foley, James; vanDam, Andries; Feiner, Steven; Hughes, John. *Computer Graphics*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.
- Bartels, Richard; Beatty, John; Barsky, Brian. *An Introduction to Splines for use in Computer Graphics & Geometric Modeling*. Los Altos, California: Morgan Kaufmann Publishers, Inc., 1987.
- Farin, Gerald. *Curves and Surfaces for Computer Aided Design*. Boston, Massachusetts: Academic Press, 1990.
- Wyszecki, G³nter. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. John Wiley & Sons, Inc., 1982.

Useful Information

- See also IGraphicText in the text display classes to create and manipulate text as a graphic.
- See also the 2DCombo sample in the code samples directory.
- The Open Class graphics classes use the following naming conventions to help you distinguish graphics classes from geometry classes:
  - Graphics class names begin with the letter 'I' followed by the name of the graphic
    (for example, ILine).
  - Geometry class names begin with the letters 'IG' followed by the name of the geometry
    (for example, IGLine2D).
- This guide describes the classes and data types common to the graphics classes. Member functions throughout the framework use some terms to indicate a particular function’s role in storage management. These terms are provided here for your convenience while reading this guide.

Functions that allocate and take responsibility for storage, or return pointers to instances, have special names and abide by the following guidelines:

- Create: When you see 'Create' in the name, the function makes and returns a new instance that the caller must delete.
- Copy: When you see 'Copy' in the name, the function copies an existing instance and the caller must delete that copy.
- Adopt: When you see 'Adopt' in the name, the function accepts an instance and is responsible for deleting it. Adopt functions that cannot have 'Adopt' in their names (for example, constructors) prefix it to the name of the formal parameter.
- Orphan: When you see 'Orphan' in the name, the function transfers responsibility for deleting an instance to the caller.
- Get: When you see 'Get' in the name, the function returns a pointer to an instance, and the caller is not responsible for deleting the instance.

2D Graphics Features

The 2D Graphics Framework provides a full range of capabilities for high-quality 2D graphics.

- IEEE double-precision (64-bit) floating point arithmetic: Arithmetic calculations are performed with 64-bit floating point data types. This gives you much finer resolution and greater accuracy.
- Graphics and Geometry classes: The framework includes a full range of extensible 2D graphics classes. The graphics functionality is separated (put in different classes) from the geometry shapes to give you more flexibility. You can use graphics and geometry classes together, or use the geometry classes by themselves.
- NURBS: The 2D curve geometries are defined by Non-Uniform Rational B-Splines (NURBS). Non-uniform means that the parameterization of the curve can be changed to allow kinks, gaps, and smooth joins. Rational means that rational polynomials are used to allow exact representations of circles, ellipses, and other conic sections. B-Spline stands for Basis spline, which is a spline curve very similar to the BÚzier curve, but provides more local control and continuity control to give you better curve fitting and modeling.
- Customizable architecture: The application-programming interfaces are clean so that you can easily extend or customize only those classes that require additional functionality and leave all other classes as they are.

Basic Concepts for 2D Graphics

This section describes basic concepts for 2D graphics:

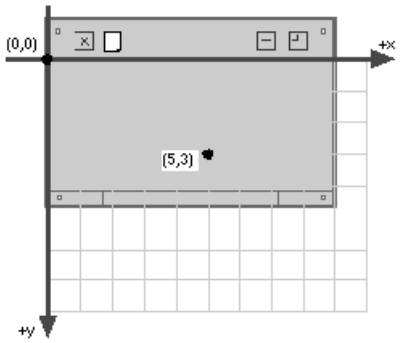
- 2D Coordinate System and Unit Resolution
- Basic Numeric Data Types

Understanding these features and mechanisms help you use the 2D Graphics framework more effectively.

2D Coordinate System and Unit Resolution

The 2D geometry classes are specified in x and y coordinate units. As shown in the figure, the default 2D coordinate system is left-handed (y-axis going down). The 2D Graphics Framework uses left-handed 2D coordinate system as a default because a typical document page starts at the top and extends downward. This is referred to as World Coordinate space through out this guide.

NOTE: The enumerated type EDeviceCoordinateSystem defined in IExtendedRootGrafPort allows you specify left or right handed coordinate system. If the kLefthand value is used, the 2D Graphics Framework uses the left-handed coordinate system. If the kRighthand value is used, the 2DGraphics Framework uses the right-handed coordinate system.



Positive coordinate units along the x-axis lie to the right of the origin and positive coordinate units along the y-axis lie below the origin. The grid lines represent where points in coordinate units lie. For example, the point (5, 3) is five grid lines in the positive x direction, and three grid lines in the positive y direction. The position of the origin (0,0) is in the upper-left corner of the view.

The resolution of any coordinate or dimension is assumed to be one pixel on a screen display. For printing and other devices, this coordinate unit is defined as 1/72 inch (one seventy-second of an inch) to equal 1 point. A point is the unit of measurement used by printers, graphic designers, and type setters. Therefore, when printing a graphic view of a one inch window on the 96 dpi CRT display, the resulting area on paper is 1.25 (one and one-quarter) inch in size, with a transform being identity because it is common and convenient for desktop publishing system to assume that 1 pixel equals 1 point. For user interface design on screen, it is essential to have exact number of pixels as the unit rather than other units.

Numeric and Arithmetic Data Types

2D geometry is based on IEEE double-precision floating point numerical values. The system has the following benefits and reduces the risk of numeric errors:

- Precision
- Speed: Many RISC processors and math co-processors work directly with double precision.
- Stability: Double precision is much more immune to round-off, under and overflow problems than fixed or single precision.

The following data types define the variables used in 2D calculations.

- GCoordinate
- GDegrees
- GParametric
- GIntensity

GCoordinate

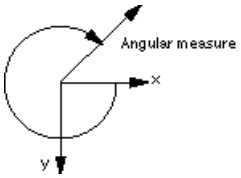
typedef double GCoordinate;

GCoordinate expresses values defined or measured in 2D coordinate units such as positions or lengths. It maps directly to pixels on the screen and logical 1/72 inch on printers. If metric fidelity is required between the screen and printer, use IMetricGrafPort. Variables of type GCoordinate are used in all of the geometry classes to allow for smooth curves. GCoordinate provides sufficient range and precision for large documents, complex curves, and matrix operations.

GDegrees

typedef double GDegrees;

GDegrees measures angles in degrees and is used when an angle needs to be specified or returned. Angles start with 0 degrees at the +x-axis, and they increase from 0 degrees to 360 degrees going clockwise in 2D (because the +y-axis points down). Angles greater than 360 degrees are interpreted as modulo 360 degrees. Negative angles are added to a multiple of 360 degrees.



GParametric

typedef double GParametric;

GParametric represents the parameter of a parametric curve, loop, or surface. The parametric curve or loop is represented by the function P(u) that returns point P on the curve for a particular value of the parameter u. P(0) returns the point at the start of the curve, and P(1) returns the point at the end of the curve

GIntensity

typedef float GIntensity;

GIntensity is a 32-bit data type that represents a range between 0 and 1.0 that is used in most device color spaces to represent color intensity. This data type is found in the IBasicColor class.

Attributes and Bundles Overview

2D Graphics attributes and bundles classes provide primitive attributes, such as paint, pen, and image transfer mode. It also provides IGrafBundle which is the collection class for combinations of these primitive attributes used to render a 2D graphic geometry. Attributes and bundles classes are mainly used for assigning and collecting attributes for 2D graphics.

Attributes are the characteristics that describe how geometries are rendered. In a graphics system, you cannot render geometry without specifying the attributes because the system has no information about how to display the geometry. You can set the attributes for a rendering device without actually rendering any geometry. You can represent that set by a collection of attribute values. IAttributeState and its subclasses are designed to fill this role.

The number of attributes required by an application depends on the purpose. The 2D Graphics Framework provides a mechanism, a bundle, to collect attributes together for rendering. The IGrafBundle class collects together all attributes needed to render a geometry.

A graphic object can have an IGrafBundle or another subclass of IAttributeState. The 2D Graphics Framework provides some commonly used IAttributeState classes.

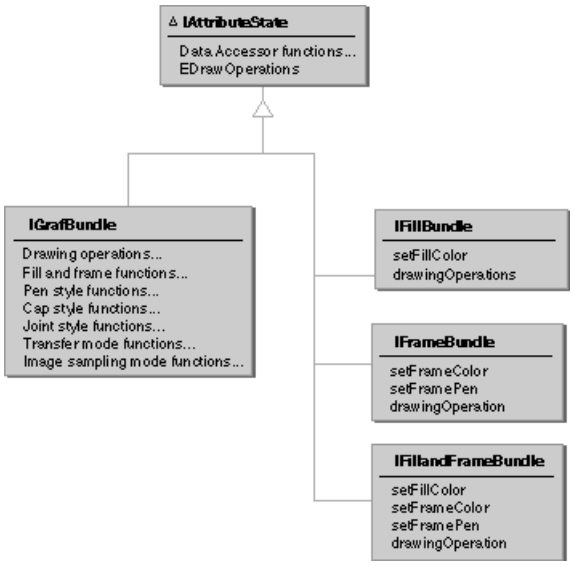
Attribute	Description
Drawing operations	Determine whether the attribute bundle is for fill, frame, or fill and frame.
Paints	Fill paint: Specifies the color of the fill for a 2D graphic. Frame paint: Specifies the color of the frame for a 2D graphic. Fill and frame paint: Specifies the color and pattern of the fill or frame for a 2D graphic.
Transfer modes	Fill transfer modes: (All 2D graphics drawn into a 2D image.) Provides algorithms to determine the color of a fill pixel when a 2D graphic fill pixel of one color overlays a 2D graphic or image pixel of another color. Frame transfer modes: (All 2D graphics drawn into a 2D image.) Provides algorithms to determine the color of a frame pixel when a 2D graphic frame pixel of one color overlays a 2D graphic or image pixel of another color. Image transfer modes: (2D images only.) Provides algorithms to determine the color of a 2D image pixel when a pixel of one color overlays a pixel of another color.
Frame styles	Pen styles: Specifies the pattern of the frame. Cap styles: Assigns a flush, square, or round shape that is added to the start or end of the frame for a line geometry to create a finished effect. Joint styles: Assigns a filler for the gap that is created in the frame by a sharp joint in a curve, using the standard bevel, round, and mitre frame joints.
Image sampling	Provides algorithms for sampling an image prior to drawing when it has been enlarged (enlarging the pixel information) or shrunk (compressing the pixel information). Algorithms make tradeoffs between quality and drawing speed.

Attribute and Bundle Classes

IAttributeState

IAttributeState is an abstract base class that defines drawing attributes for 2D graphics.

The figure shows the 2D attribute bundle classes derived from the abstract base class IAttributeState. IAttributeState has pure virtual data accessor functions to retrieve information about the attributes and drawing operation for the bundle. The drawing operation determines whether the bundle provides fills, frames, or fills and frames.



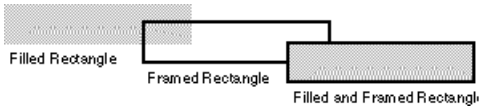
IGrafBundle

IGrafBundle is a collection of attributes that contains imaging information for the graphic rendering device. IGrafBundle is derived from IAttributeState and can include:

- Fill and frame attributes
- Image attributes

Fill and Frame Attributes

The fill is a color that spreads over the interior of the geometry, and the frame is the color and line that outlines the border of the geometry. The draw operation tells the rendering device to fill the geometry, frame the geometry, or fill and frame the geometry, as shown in the figure:



The most frequently used attributes are fill paint, frame paint, pen, and draw operations. These attributes each have their own constructors. IPaint is the class

used for specifying the fill paint and the frame paint. EDrawOperation is an enumerated type, defined in IAttributeState, used to specify frame only, fill only, or fill and frame operations for a given geometry. The following example specifies frame only:

```
IGrafBundle(const IPaint& fPaint,
      IAttributeState::EDrawOperation attribute = kFrame);
```

If attribute equals kFrame, the paint is for framing; if attribute equals kFill, the paint is for filling; and if attribute equals kFillAndFrame, the paint is used for both fill and frame. The following example specifies different paint for filling and framing respectively:

```
IGrafBundle(const IPaint& fillPaint,
      const IPaint& framePaint,
      IAttributeState::EDrawOperation attribute = kFillAndFrame);
```

The IBaseColor class specifies the color in IPaint. IBasic color can construct IPaint as a parameter. In the following example, constructors take IBaseColor as a parameter instead of referencing instances of IPaint:

```
IGrafBundle(const IBaseColor& fillColor,
      const IBaseColor& frameColor,
      IAttributeState::EDrawOperation attribute = kFillAndFrame);
```

NOTE: The constructors in this example have the same semantics as the previous constructors except that IPaint is created by the constructor on the default heap.

Functions are available to add, remove, and change the standard fill, frame and draw attributes. The function for the drawing operations attribute is as follows:

```
virtual IAttributeState::EDrawOperation drawingOperation() const;
virtual void setDrawingOperation(IAttributeState::EDrawOperation);
```

The function for the fill paint attribute is as follows:

```
virtual const IPaint* fillPaint() const;
virtual void setFillPaint(const IPaint& paint);
```

The function for the frame paint attribute is as follows:

```
virtual const IPaint* framePaint() const;
virtual void setFramePaint(const IPaint& paint);
```

For convenience, in addition to the functions for fill and frame, the IPaint attribute provides the following data accessors of the IBaseColor fill and frame elements:

```
virtual const IBaseColor* fillColor() const;
virtual void setFillColor(const IBaseColor& color);
virtual const IBaseColor* frameColor() const;
virtual void setFrameColor(const IBaseColor& color);
```

The function for the fill paint transfer mode attribute is as follows:

```
virtual const IColorTransferMode* fillTransferMode() const;
virtual void setFillTransferMode(const IColorTransferMode&);
```

The function for the frame paint transfer mode attribute is as follows:

```
virtual const IColorTransferMode* frameTransferMode() const;
virtual void setFrameTransferMode(const IColorTransferMode&);
```

The function for the frame pen attribute is as follows:

```
virtual const IPen* framePen() const;
virtual void setFramePen(const IPen&);
```

The function for the frame end cap attribute is as follows:

```
virtual const ICap* frameEndCap() const;
virtual void setFrameEndCap(const ICap&);
```

The function for the frame joint attribute is as follows:

```
virtual const IJoint* frameJoint() const;
virtual void setFrameJoint(const IJoint&);
```

Image Attributes

IGrafBundle has the following functions to add, remove, and change the standard image attribute.

The function for the image transfer attribute is as follows:

```
virtual const IImageTransferMode* imageTransferMode() const;
virtual void setImageTransferMode(const IImageTransferMode&);
```

The function for the image sampling control attribute is as follows:

```
virtual const IImageSamplingControl* imageSampling() const;
virtual void setImageSampling(const IImageSamplingControl&);
```

The image transfer mode and the image sampling control apply for the geometry of both IGImage and the IGImage set in IPaint for image pattern fill.

Special Purpose Attribute Bundles

While IGrafBundle provides more functionality than the special purpose attribute bundles by allowing you to specify transfer mode and image sampling settings for 2D image graphics, special purpose bundles are still provided. These classes are easy to construct and use. The supported special purpose attribute bundle classes are IFillBundle, IFrameBundle, and IFillAndFrameBundle.

IFillBundle

The color is applied to the fill only, and one constructor lets you pass a transfer mode that is applied to the fill color.

```
IFillBundle (const IBaseColor& fillColor);
IFillBundle (const IBaseColor& fillColor, const IColorTransferMode& transferMode);
```

IFrameBundle

These constructors have default pen width and pen balance values. The color is applied to the frame only and one constructor lets you pass a transfer mode that is applied to the frame color.

```
IFrameBundle ( const IBaseColor& frameColor,
    GCoordinate penwidth = 1,
    IPen::EPenBalance balance = IPen::kCenteredFrame );

IFrameBundle ( const IBaseColor& frameColor,
    const IColorTransferMode& transferMode,
    GCoordinate penWidth = 1,
    IPen::EPenBalance balance = IPen::kCenteredFrame );
```

IFillandFrameBundle

These constructors have default pen width and pen balance values. The colors are applid to the fill and frame, and one constructor lets you pass two transfer modes. One transfer mode is applied to the fill color and the other is applied to the frame color.

```
IFillandFrameBundle ( const IBaseColor& fillColor,
                      IBaseColor& frameColor,
                      GCoordinate penwidth = 1,
                      IPen::EPenBalance balance = IPen::kCenteredFrame );

IFillandFrameBundle ( const IBaseColor& fillColor,
                      const IColorTransferMode& fillTransferMode,
                      IBaseColor& frameColor,
                      const IColorTransferMode& frameTransferMode,
                      GCoordinate penWidth = 1,
                      IPen::EPenBalance balance = IPen::kCenteredFrame );
```

Color and Pattern Attributes

Color and Pattern Attributes classes specify color appearances and spaces, and allow you to create or modify existing patterns.

IPaint

The IPaint attribute specifies the color appearance of the geometry source primitive. The 2D Graphics Framework differs from traditional graphics systems in that it does not use a mechanism for foreground color, foreground pattern, background color, or background pattern. Instead, you specify a fill and frame color appearance by using a combination of IPaint objects and a transfer mode attribute.

The IPaint class is a concrete class that encapsulates the color and image pattern you can set. If you do not specify a color or image pattern, IPaint provides a default value.

```
class IPaint {
public:
    IPaint ();
    virtual ~IPaint ();
```

This constructor takes a reference of a color of type IBaseColor and a mask pattern of type IMaskPattern as parameters:

```
IPaint(
    const IBaseColor& aColor,
    const IMaskPattern& maskPattern = IMaskPattern::kSolid,
    const IGPoint2D& patternPhase = IGPoint2D::origin());
```

The patternPhase parameter defines the offset for tiling the pattern. The default value is the origin of the object. This parameter is also called patternOrigin in Visual Age C++ 3.5.

This constructor takes a reference of an image pattern of type IGImage as a parameter:

```
IPaint (
    const IGImage& imagePattern);
    const IGPoint2D& patternPhase = IGPoint2D::origin());
```

The IPaint attribute provides the following data accessors of the IBaseColor color element:

```
virtual void setColor(const IBaseColor& color);
virtual const IBaseColor* color() const;
```

The IPaint attribute provides the following data accessors of the IMaskPattern mask pattern element:

```
virtual const IMaskPattern* maskPattern() const;
virtual void setMaskPattern(const IMaskPattern& MaskPattern =
IMaskPattern::kSolid);
```

The IPaint attribute provides the following data accessors of the IGImage image pattern element:

```
virtual void setImagePattern(const IGImage& imagePattern);
virtual const IGImage* imagePattern() const;
```

To support the polymorphic subclass in the ease-of-use copy semantic API of the IGrafBundle use:

```
StreamableDeclarationMacro(IPaint);
```

Each IPaint attribute contains an instance of IBaseColor. IPaint also has the image pattern which allows for the specification of a repeating image as the filled or framed appearance. Each IPaint instance contains a pointer to an instance of an IGImage.

WIN If the IGImage is larger than 8x8 pixels, only the upper left corner of the image is used.

To extend the capability of IPaint, you can subclass IPaint to add attributes that allow you to express, for example, gradients, fills, or 2D shaders.

WIN Equivalent functions are supported on GDI as 'Bitmap as Brushes' functions.

IBaseColor

IBaseColor is a concrete class that encapsulates all possible color spaces including RGB, HSV, and CIE-XYZ. It has the default representation of compact RGB color (where each element of red, green, and blue is represented by an 8-bit unsigned number). The IBaseColor class is as follows:

```
class IBaseColor {
public:
```

IBaseColor uses two data types to define color ranges and intensities:

- GIntensity
- CharIntensity

GIntensity is of type float. CharIntensity is an unsigned character.

The enumerated type EPredefinedColor defines some commonly used colors:

- kWhite
- kBlue
- kRed
- kPink
- kGreen
- kCyan
- kYellow
- kBlack
- kDarkGray
- kDarkBlue
- kDarkRed
- kDarkPink
- kDarkGreen
- kDarkCyan
- kBrown
- kPaleGray

The default constructor sets default color to black:

```
IBaseColor();
virtual ~IBaseColor();
```

The following constructor takes the enumerated type EPredefinedColor as a parameter.

```
IBaseColor(EPredefinedColor commonlyUsedColor);
```

The following constructor takes gray intensities as parameters:

IBaseColor(CharIntensity gray, CharIntensity opacity = 255);

The following constructor takes RGB colors as parameters. The precision is only 8 bit by default, but you can subclass it to enhance precision.

IBaseColor(CharIntensity red, CharIntensity green, CharIntensity blue, CharIntensity opacity = 255);

IBaseColor provides the following data accessors for default compact RGB color support:

```
CharIntensity red() const;
CharIntensity green() const;
CharIntensity blue() const;
CharIntensity opacity() const;
virtual void setRed(CharIntensity red);
virtual void setGreen(CharIntensity green);
virtual void setBlue(CharIntensity blue);
virtual void setOpacity(CharIntensity opacity);
```

IBaseColor provides the following assignment operator, which subclasses must override:

```
virtual IBaseColor& operator=(const IBaseColor&);
```

To support the polymorphic subclass in the ease-of-use copy semantic API of the IGrafBundle use:

```
StreamableDeclarationMacro(IPaint);
```

For conversion to device-independent universal CIE-XYZ color space, your subclass should implement the following conversion functions that provide access for color conversion and color matching capabilities:

```
virtual void elementsInXYZColorSpace(
    GIntensity& X,
    GIntensity& Y,
    GIntensity& Z) const;
virtual void setElementsInXYZColorSpace(
    GIntensity X,
    GIntensity Y,
    GIntensity Z) const;
```

NOTE: GIntensity is used here for precision calculation. It is defined to be a single-precision floating-point number.

For better precision than the default compact RGB color representation, your subclass can override these virtual member functions where precision is required. The base class implementation only supports the compact RGB color.

You can subclass IBaseColor to represent color metric in different precision and different color space. Your subclass must override the assignment operator and these virtual member functions:

```
virtual red(GIntensity& red) const;
virtual green(GIntensity& green) const;
```

```
virtual blue(GIntensity& blue) const;|
virtual void setRed(GIntensity red);
virtual void setGreen(GIntensity green);
virtual void setBlue(GIntensity blue);
```

The subclass constructor must at least initialize the base class by calling the base class constructor that takes RGB color as the parameter as follows:.

```
IBaseColor(
    CharIntensity Red,
    CharIntensity Green,
    CharIntensity Blue);
```

You can also do this by calling the setters in the base class. Otherwise the default compact RGB color support does not represent any specific color other than the default black. In that case, the color specified in the subclass is wrong because it differs from what appears in the base class. The subclass can override all virtual functions in the base class, thus overloading the default compact RGB color support.

IMaskPattern

IMaskPattern is an 8-by-8 image pattern of 1-bit color depth (black = transparent, white = solid). You can create the pattern or use one of the supported patterns defined by the enumerated type EPattern:

- kSolidPattern
- kDiagonalDownPattern
- kCrossPattern
- kDiagonalCrossPattern
- kDiagonalUpPattern
- kHorizontalPattern
- kVerticalPattern

When creating an image pattern, the pre-defined IMaskPattern provides a transparent background and uses the IPaint color specified to draw the foreground. However, for a custom IMaskPattern, the background is opaque and the background color will always be the default color (black). The foreground color for the custom IMaskPattern should still be the IPaint color. In short, a user-defined IMaskPattern will always have a black background, and only a pre-defined IMaskPattern will have a transparent background.

Frame Styles

The frame of the geometry is drawn using the following attributes: IPaint, IColorTransferMode, IPen, IJoint, and ICap. IPaint and IColorTransferMode apply to both the frame and the inside of the geometry while IPen, IJoint, and ICap define the frame of the geometry.

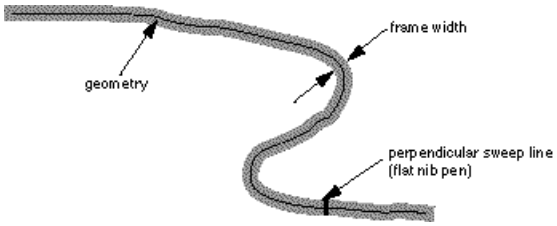
This section describes the classes that define frame styles of the geometry.

IPen

IPen is a concrete class that describes how to frame the edge of a geometry. IPen defines a pen by a pen width, pen balance, and pen type. By default, it is a centered-solid pen.

Pen width

Width is a floating-point value that represents the device-independent width of the pen in coordinate units. A centered-frame pen of a width can be thought of as using a pen with a flat nib of the specified pen width, and drawing a curve with the nib perpendicular to the direction of the curve with the center of the nib on the curve. In the figure, the 'perpendicular sweep line' is the flat nib:



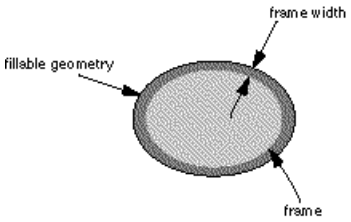
Balance

The enumerated type EPenBalance has the following values:

- kCenteredFrame
- kInsetFrame

For kCenteredFrame, the pen is centered over the border of the geometry for path and area-enclosing geometries.

For kInsetFrame, the pen fits exactly along the inside edge of a fill and covers the edge-fill pixels of a filled area-enclosing geometry as shown in the figure:



The kInsetFrame value is useful for specifying the outside edge of a frame with pixel perfection rules. The kInsetFrame value describes a pen (of the pen width specified in the base class) on the inside of the edge of the geometry. The kInsetframe value is only used for geometries that can be filled. When a geometry object to frame is not fillable, the pen behaves as a center framed IPen. An IPen kInsetFrame applied to IGLine2D, IGPolyline2D, and IGCurve2D always behaves as a center framed IPen.

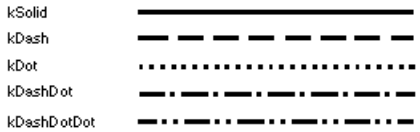
Pen Types

The enumerated type EPenType defines the values of different pen styles:

- kHairline
- kSolid
- kDashed
- kDot, kDashDot
- kDashDotDot
- kInvisible
- kUnknownPen

NOTE: The simplest pen style is kHairline. The kHairline style means the device renders a geometry frame using the fastest possible means. End caps, joints, and transfer modes are not applied to an IPen kHairline. Use the kHairline style only when speed is more important than accuracy. The only constraint placed on a hairline pen is that it must render within 1 pixel from the edge of a geometry.

The following figure shows some of the pen styles available:



NOTE: The layout of the patterns of these pens is not predictable because the underlying platform support is not consistent.

WIN OS/2 The kDashed, kDot, kDashDot, and kDashDotDot pen styles work only when pen width is one or less.

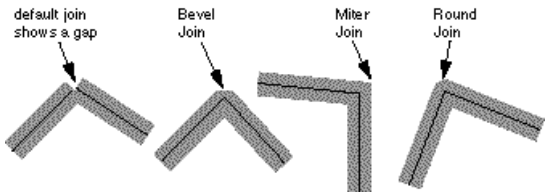
IJoint

The joint style controls the rendering of sharp points. Every sharp point causes a gap to occur between the edges of the frame
(C1 discontinuity). The IJoint class determines how to fill the gap of a sharp joint.

The enumerated type EJointType defines the standard joint styles as follows:

- kBevel
- kMiterLimit
- kRound
- kUnknownJoint.

The following figure shows the standard joint styles:



The bevel joint style (kBevel) simply closes the gap between one edge and the next with a triangle.

The miter joint style (kMiterLimit) takes the two pens, and continues running in the original directions until the outside edges meet. The miter joint style has limitations. As the angle of the joint becomes greater, the limitation of what the joint can become when rendered is not predictable. On Windows platforms, it is dependent on the GDI device drivers.

The round joint style (kRound) puts a conic section in the gap such that tangent continuity is maintained.

ICap

ICap places a cap at the start, at the end, or at an interior discontinuity (C0 discontinuity) of the frame of a linear geometry (IGLine2D, IGPolyline2D, IGCurve2D). Caps are not used on area geometries (IGRect2D, IGEllipse2D, IGPolygon2D, IGLoop2D, IGArea).

The enumerated type ECapType defines the supported cap styles:

- kFlush: A flush end cap does nothing.
- kSquare
- kRound
- kUnknownCap

The figure shows the supported cap styles:



A flush end cap (kFlush) does nothing.

A square cap (kSquare) extends the curve in a straight line by half the pen width.

A round cap (kRound) puts a semi-circle at the start or end of the curve.

Transfer Modes

A transfer mode is the specification of an interaction that takes place on the color of the primitive being drawn, on the current color at the destination location, or between both of these colors.

Transfer modes are algorithms that determine the resulting color when pixels of one color (source color) overlay pixels of another color (destination color) when 2D graphics are drawn. There is a wide range of transfer modes so that you can blend colors to achieve different visual effects when you work with 2D images.

One IGrafBundle can own a transfer mode for a fill, frame, and image. The transfer mode adopted for fill and frame applies to the 2D graphic that owns that bundle

only when that graphic is drawn to a 2D image. The transfer mode adopted for an image applies to a 2D image when it is drawn to a view or when one 2D image is drawn on top of another 2D image.

Most transfer modes operate within a specific color space, which makes transfer modes device independent. When no transfer mode is adopted, the source color is completely copied over the destination color.

For information on mode type definitions, see texts on Windows graphics systems.

IColorTransferMode

The color transfer mode applies for color paint of fill and frame for the geometry to be rendered. In the case of rendering a geometry of IGImage or the IGImage set in the IPaint for image pattern fill, the color transfer mode implies the color is applied using such mode to the area the IGImage is rendered.

The default behavior of IColorTransferMode is source-replaces-destination or source copy mode.

The enumerated type EColorMode describes a set of supported color transfer modes:

- kSourceCopy
- kInvertSource
- kDestinationCopy
- kInvertDestination
- kOR
- kAND
- kXOR
- kInvertedSourceAND
- kInvertedSourceOR
- kInvertedDestinationAND
- kInvertedDestinationOR
- kInvertedAND
- kInvertedOR
- kInvertedXOR
- kONES
- kZEROS

IImageTransferMode

The image transfer mode applies for the geometry of IGImage. The default behavior of IImageTransferMode is source-replaces-destination or source copy.

The enumerated type EImageMode describes the supported set of image transfer modes:

- kSrcCopy
- kSrcORDst
- kSrcANDDst
- kSrcXORDst

- kSrcANDInvertDst
- kInvertSrc
- kInvertResultOfSrcORDst
- kSrcANDPattern
- kInvertSrcORDst
- kPatternCopy
- kPatternORInvertSrcORDst
- kPatternXORDst
- kInvertDst
- kAllZeros
- kAllOnes

Image Sampling

Image sampling is the process of investigating the status of an image prior to drawing it when the image is either:

- Interpolated (enlarged), increasing the number of pixels during fractional translates and scales that result in stretching
- Decimated (shrunk), reducing the number of pixels during scale operations that result in shrinking

The image sampling control applies for the geometry of IGImage only. When a rendered image undergoes a transformation composed of something other than an integral translation, it is necessary to resample the image. When an image is drawn on one device and then drawn on another device of a different resolution, it is interpolated or decimated as appropriate to maintain the same size on both devices. Interpolation and decimation also occur during rotate, scale, perspective, and translate operations on the image graphic. Both interpolation and decimation can be used during a perspective transformation.

The IImageSamplingControl attribute allows you to control the filtering method employed during interpolation and decimation.

The enumerated type ESampleMode provides several stock filtering methods:

- kBlackOnWhite
- kWhiteOnBlack
- kColorOnColor
- kHalfTone

WIN These modes map directly into corresponding StretchBltMode for the GDI StretchBlt function.

Chapter 18. 2-D Graphics Geometry

2D Graphics Geometry Overview

The 2D Graphics framework provides geometry classes and corresponding IMGraphic-derived graphics classes. The classes described here are the basic geometric forms that the 2D Graphics framework knows how to render based on the underlying geometry classes, the 'atoms' for building more sophisticated graphics.

Geometry Classes

The 2D Graphics framework geometry classes fall into the following general categories:

- Computational geometry - basic definitions for doing geometric computations. Computational geometry is used for calculations. It is not rendered directly except IGRect2D, which is filled and framed like other area-enclosing geometry). Use these classes for constructing other types of geometry.
- Path geometry - primitives that divide space. Path geometry includes lines, polylines and curves. They may be framed, but not filled.
- Area geometry - primitives enclosing space. They may be framed and filled.
- Special area geometry - classes for making area geometry by concatenating different geometric shapes, used mostly for clipping.
- Image geometry - classes for image handling and manipulation.
- Path geometry classes describe open shapes that need not be filled. Area geometry classes however describe shapes of enclosed path that can be filled. Some geometric shapes such as point, line and non-enclosing polyline and non-enclosing curves are of path geometry while enclosing geometric shapes such as rectangle, circle, ellipse, polygon and enclosing curves (loops) can be both path and area geometry.

Graphic Classes

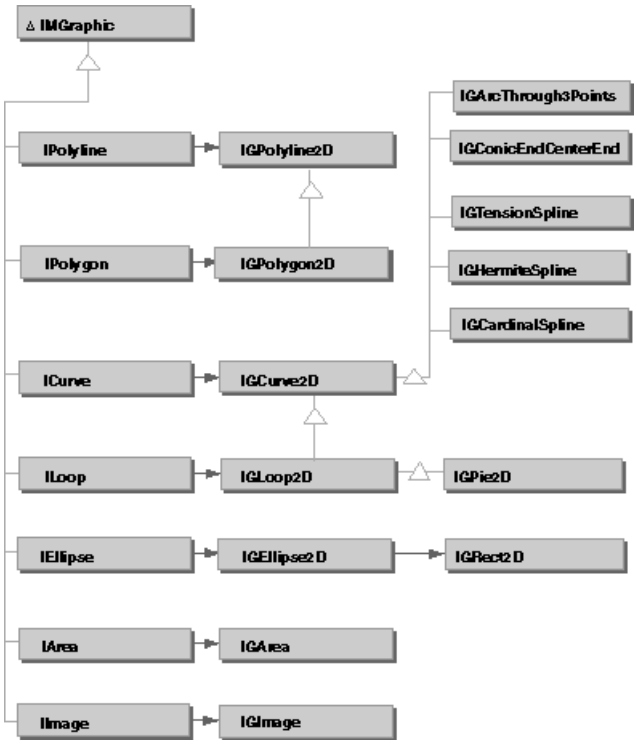
The table summarizes the graphic classes, derived from IMGraphic, and their corresponding geometry classes:

Graphic Class	Geometry Class
ILine	IGLine2D: A line segment formed by 2 points.
ICurve	IGCurve2D: A general purpose approximating spline curves used for arc and other quadratic curves.
IPolyline	IGPolyline2D: A set of connected line segments.
ILoop	IGLoop2D: A closed curve.
IEllipse	IGEllipse2D: Circles and ellipses.
IPolygon	IGPolygon2D: A polygon with an arbitrary number of 2D vertices.
IArea	IGArea: A set of geometries enclosing areas combined by boolean operations.
IImage	IGImage: A pixel-based image.

© Copyright IBM Corp. 1998

Graphics Class Architecture

The 2D graphics classes whose names specify a geometry (ILine, ICurve, IEllipse, and so on) are derived from IMGraphic. As shown in the figure, a 2D graphics class has a geometry class that provides the shape for the graphic.



The IMGraphic-derived graphics classes are wrappers for their referenced geometry classes. The graphics classes have functions that let you manipulate the geometry from within the graphic without accessing the geometry functions directly. The advantage of the wrapper architecture is that the 2D graphics classes bring together the functionality of the geometry classes and the high-level functionality of the IMGraphic class.

All IMGraphic-derived classes use a drawing port (IGrafPort) to render the graphic; they own an attribute bundle (IGrafBundle); and they know the transformation matrix.

Computational Geometry Classes

Computational geometry classes consist of basic definitions for completing geometric computations. Computational geometry is used for calculations. It is not rendered directly except IGRect2D, which is filled and framed like other area-enclosing geometry. Use these classes for constructing other types of geometry.

Points

The IGPoint2D class defines (x, y) coordinates that represent a point or vector on the 2D coordinate plane. The IGRPoint2D class defines (x, y, w) coordinate values that represent a homogeneous (rational) point or vector on the 2D coordinate plane. The IGPoint2D and IGRPoint2D member functions are summarized here:

IGPoint2D:

- crossProduct
- dotProduct
- interpolate
- setPolarCoordinates
- setVectorLength
- setVectorAngle
- vectorLength
- vectorAngle

IGRPoint2D

- divW
- dropW

IGPoint2D

IGPoint2D defines points on the coordinate plane. Some functions also treat IGPoint2Ds as vectors. IGPoint2Ds are used as the basic building blocks for most of the geometric primitives, and as a computational type for many types of geometric calculations.

This is a primitive type, and is not intended to be subclassed. IGPoint2Ds are treated much like numbers, with the typical operators (+, -, *, /) defined for them and acting component-wise on the fX and fY variables.

For example,

```
IGPoint2D a, b, c;
c = a + b;
```

Is equivalent to:

```
IGPoint2D a, b, c;
c.fX = a.fX + b.fX;
c.fY = a.fY + b.fY;
```

The complete list of operators defined for points is:

+, -, *, /, +=, -=, *=, /=, ==, !=, []

A numeric value can be supplied with the * and / operators to work as a scalar multiply; (for example, it is treated as a point with both fX and fY set to the value). The [ ] operator indexes the components (p[0] returns fX, p[1] returns fY). IGPoint2D's member variables, fX and fY are public, and designed to be directly accessed.

Mathematics makes distinction between points and vectors. Because their representation is the same, the 2D Graphics framework uses IGPoint2D in both roles.

IGRPoint2D

IGRPoint2D defines rational points containing a third coordinate, fW. They are most commonly used for defining rational or homogeneous coordinates in IGCurve2D. Curves use the rational coordinates to accurately represent conic sections (curves, arcs, ellipses, and so on). Rational points are also occasionally useful for situations where you need to keep track of a 'perspective' term. Typically, you do not deal directly with IGRPoint2Ds, because the curve and transformation implementation manage the mathematical details.

Unlike IGPoint2D, the only operators defined for IGRPoint2D are ==, != and []. The [] operator indexes the fX, fY and fW components (p[0] returns fX, p[1] returns fY, and p[2] returns fW).

You can construct IGRPoint2Ds directly from IGPoint2Ds. In this case the value of the rational component fW is set to 1.0. Conversion from IGRPoint2Ds to IGPoint2Ds must be done explicitly via the divW and dropW methods.

IGRect2D

The IGRect2D class defines an axis-aligned rectangle defined by four coordinates, the top, left, bottom and right. Like IGPoint2D, IGRect2D is a 'primitive' type, and not intended to be subclassed or extended. Also like IGPoint2D, the member variables of IGRect2D are public for ease of access. An IGRect2D is renderable by the IGrafPort as both a filled and a framed object.

When IGrafPort renders a filled and framed rectangle, the frame occupies the exterior boundary of the rectangle and the filled area occupies the entire rectangle except for the right and the lower exterior boundaries. For example, the frame for an IGRect2D with the dimensions (0,0) - (5,5) consists of four lines running from (0,0) to (5,0) to (5,5) to (0,5) to (0,0). The filled area for the rectangle occupies all the elements from (0,0) to (4,0), (0,1) to (4,1), down to and including (0,4) to (4,4). Thus, the filled area overlaps the upper and left frame lines, but not the lower and right frame lines.

An IGRect2D is considered to be 'empty' if fRight <= fLeft or fBottom <= fTop. Empty IGRect2Ds do not fill, and the extendTo ignores them. To find out if a rectangle is empty, call the isEmpty function.

Point Arrays

IGPoint2DArray handles arrays of IGPoint2Ds and is the basis for IGPolyline2D and IGPolygon2D. IGRPoint2DArray handles arrays of IGRPoint2Ds and is the basis for IGCurve2D and IGLoop2D). The point array member functions are summarized here:

IGPoint2DArray

- bounds
- resize

IGRPoint2DArray

- numberofPoints
- point
- setPoint
- swap

IGPoint2DArrays are used to store and manipulate linear arrays of points (IGPoint2D and IGRPoint2D). They are designed to be efficient, compact and simple. These arrays are subclasses of IRawArray templates instantiated with the appropriate points. IRawArray does not call the constructor for each element of an array, thus improving performance.

You can create rational IGRPoint2DArray from an IGPoint2DArray (with all the w's defaulting to 1.0). The reverse is not true, however, because an explicit projection is required.

Path and Area Geometry

Path and area geometry contains primitives that divide space. Path geometry includes lines, polylines and curves. They may be framed, but not filled.

IGline2D

The IGLine2D class defines line segments that are defined by two endpoints with an implied direction that moves from the start point to the endpoint.

IGCurve2D

The IGCurve2D class is the general curve primitive in the 2D Graphics framework. Use this to implement free-form curves, arcs, and path geometry containing discontinuities. The IGCurve2D class can represent the geometry of all other primitives (lines, polygons, ellipses).

The curve classes let you implement free-form curves, arcs, and path geometry that contain discontinuities (breaks or kinks in the smoothness of the curve) and that can be joined to form long, continuous curves. IGLoop2D derives from IGCurve2D. IGCurve2D and IGLoop2D can represent the geometry of all other 2D geometry classes.

Curves are defined by Non-Uniform Rational B-Splines (NURBs). Non-uniform means that the parameterization of the curve can be changed to allow kinks, gaps, and smooth joins. Rational means that rational polynomials are used to allow exact representations of circles, ellipses, and other conic sections. B-Spline stands for Basis spline, which is a spline curve very similar to the BÚzier curve, but provides more local control and continuity to give you better curve fitting and modeling.

IGCurve2D has derived classes, listed in the table, that you can use to instantiate a IGCurve2D. These classes serve to clarify different types of arcs that have similar arguments.

Class	Description

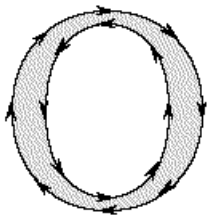
IGConicEndCenterEnd	This class creates a conic curve from a start point, endpoint, center point, and a Boolean value. The points create an elliptical shape. The conic curve tangents are perpendicular to the vectors between p0 and centerPoint, and centerPoint and p2. When the vectors are equal, a circular arc is generated that is less than 180 degrees.
IGCardinalSpline	This class creates a parametric curve that interpolates a set of control points passed to it in the array. The fit is accomplished by interpolating the middle two control points and finding the tangent at those points that are parallel to the line between the previous control point and the next control point. The tension parameter controls how tightly the curve interpolates the control polyline. The tension parameter values range from 0.0 to 1.0 where 0.0 is very tight and 1.0 is very loose. The number of points in the array must be a multiple of four.
IGArcThrough3Points	This class creates an arc that passes through three points.
IGHermiteSpline	This class generates a parametric spline by interpolating between the two endpoints passed to it in the array and their tangents. The number of points in the array must be a multiple of four.
IGTensionSpline	This class creates a parametric spline from an array of points that are interpolated by tension and bias values. The tension controls how close the curve is to the control polyline. The number of points in the array must be a multiple of four.

IGLoop2D

IGLoop2D is a subclass of IGCurve2D that draws filled shapes. All of the functions supplied for IGCurve2D also apply for IGLoop2D. IGPie2D, a subclass of IGLoop2D, creates a pie-shaped wedge.

Loops are closed curves for drawing filled curves. Loops derive from curves. IGCurve2D and IGLoop2D can represent the geometry of all other 2D geometry classes because they have constructors that accept any 2D geometry. To fill a loop, use one of the fill algorithms described in 'Filling loops and polygons.'

Discontinuities (breaks) in the curves of a loop are closed when the loop is drawn. For hit detection, the direction of each loop segment is used to determine whether a point is inside or outside the loop. For example, the letter 'O' can be created from two concentric circles with the inner circle running in the opposite parametric direction from the outer circle. This way, points that lie between the two circles are found to be on the letter.



IGEllipse2D

The ellipse classes let you implement elliptical shapes that are inscribed inside a bounding rectangle. The bounding rectangle can be passed as the following:

- An IGRect2D.
- An IGEllipse2D.

- Dimensions for a center and radius.
- Dimensions for a center, height, and width.

The enclosed ellipse touches the midpoint of each edge of the bounding rectangle. When the height and width of the bounding rectangle are equal, the enclosed ellipse is a perfect circle. IEllipse2D has an internal matrix for transforming the ellipse without altering the original points of the ellipse. When you call a transformation function on an ellipse, the values are concatenated to the matrix.

IGPolygon2D

A polygon is a set of points that are connected by straight line segments and closed automatically. Polygons derive from polylines. You should not specify the same starting and ending points to close the polygon because the last point gets connected to the first point automatically. Polygons are closed line segments for drawing filled shapes. The polygons use the fill algorithms described in 'Filling loops and polygons ' for filling the polygon.

Special Geometry

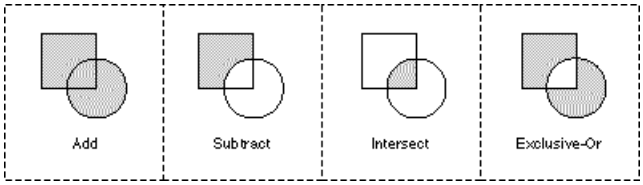
Special Geometry classes for making area geometry by concatenating different geometric shapes, used mostly for clipping.

IGArea

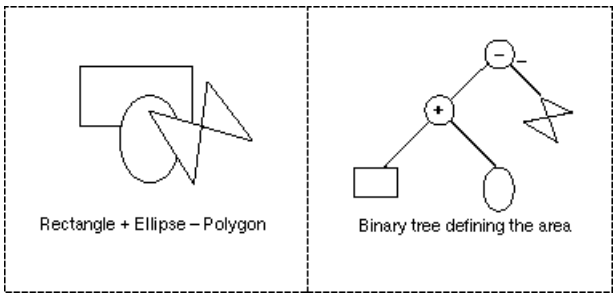
The IGArea class is a device-independent specification of an arbitrarily-shaped area. An IGArea object (henceforth referred to as an area) is defined by performing binary constructive area geometry (CAG) operations on other area-defining geometries, such as rectangles, ellipses, and polygons.



The operations are shown below.



You can form a binary tree to describe the content of any area. For example, the following area is defined to be the rectangle plus the ellipse minus the polygon.



When the area above is filled, only the highlighted outline is filled. Similarly, when it is framed, the frame is drawn around the boundary of the filled area.

Special Image: IGImage

IGImage is a special geometry. Even though it is a raster image, it does have geometric properties such as bounds and is transformable (by resampling if needed).

An image is a device-independent, pixel-based graphic that lets you work in pixels. You create an image by drawing graphics into it. The graphics are converted from coordinate units to pixels.

IGImage also encapsulates image data. Image data is normally stored in different file formats. These formats include bitmap(.BMP), GIF, PCX, and TIFF.

Chapter 19. 2-D Graphics Modeling

Modeling Overview

The 2D Graphics framework provides two distinct ways of rendering geometries on a device:

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry via a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

2D Modeling classes also provide a convenient way to associate geometry with attributes and transforms for rendering and storage.

IMGraphic is the abstract base class for representing the 2D primitives of the 2D Graphics framework. It is a higher-level manifestation of graphical objects which can be held in a collection, transformed, and rendered to a graphic device through an IGrafPort.

IMGraphic can be considered as a utility class for applications to hold geometry-related data that includes the geometry definition, its IGrafBundle (set of graphical attributes defining the representation of the geometry), and a set of transformation functions. You can derive from default IMGraphic objects and store other information specified.

Each IMGraphic object provides the following functionality:

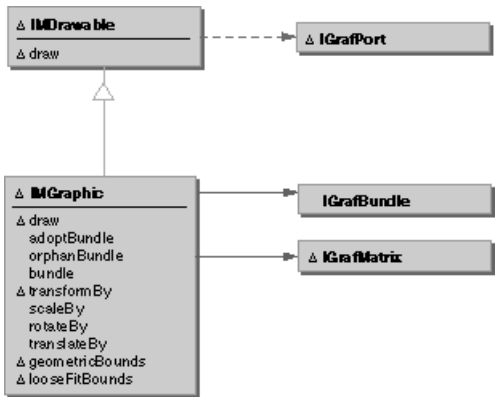
- Holds a set of its own attributes and transformations.
- Provides streaming capability (with some restrictions on some of its subclasses).
- Provides inquiry functions as a mechanism for direct manipulation of IMGraphic objects such as hit detection.
- Provides extensibility through subclassing.
- Allows, through the IGraphicGroup subclass, creation of hierarchies of IMGraphic objects. IGraphicGroup creates a collection of IMGraphic objects forming a group. Each of the IMGraphic objects can be an IGraphicGroup itself, so one can create a hierarchy of objects.

Create your own IMDrawable and IMGraphic derived class to create complex graphics such as an automobile or human figure. A complex graphic is actually a combination of the geometry classes. The drawing port can render it because it knows all of the geometry classes.

Graphics Class Architecture

The 2D graphic classes derive from IMDrawable. IMDrawable specifies the minimal protocol for drawing and streaming a 2D graphic. As shown in the figure, IMGraphic is an abstract base class that derives from IMDrawable:

© Copyright IBM Corp. 1998



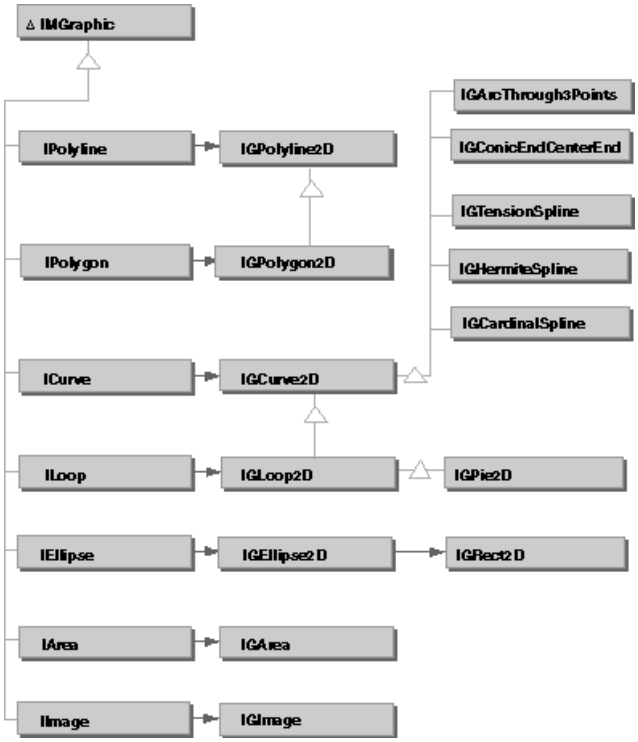
IMDrawable provides the protocol for drawing a 2D graphic. IMGraphic provides the protocol common to all 2D graphics obtaining attribute information which it uses at rendering time, transforming the graphic, and finding the bounds of the graphic.

All IMGraphic derived classes use a drawing port (IGrafPort) to render the graphic; own an attribute bundle (IGrafBundle); and know the transformation matrix for a graphic (IGrafMatrix).

All IMGraphic derived classes know how to draw themselves. The graphics provide the geometry, attribute, and transformation information to the drawing port when the draw function of the graphic is called. This is a pure virtual function for IMGraphic, and all IMGraphic subclasses have to override this functionality.

IMGraphic Class Architecture

The 2D graphic classes whose names specify a geometry, such as ILine, ICurve, IEllipse, and so on, are derived from IMGraphic. As shown in the figure, a 2D graphics class has a geometry class that provides the shape for the graphic. A graphics class has public functions specific to that graphic that allow the graphic to interact with its referenced geometry.



The graphics classes are wrappers for their referenced geometry classes. The graphics classes have functions that let you manipulate the geometry from within the graphic without accessing the geometry functions directly. The advantage to the wrapper architecture is that the 2D graphics classes bring together the functionality of the geometry classes and the high-level functionality of the IMGraphic class.

All IMGraphic-derived classes use a drawing port (IGrafPort) to render the graphic; own an attribute bundle (IGrafBundle); and know the transformation matrix for itself.

IMGraphic Models

This section provides information on IMGraphic subclasses, attributes and bundles, transformations and bounds.

IMGraphic Subclasses

To provide application specific behavior, you can create concrete IMGraphic subclasses. The following are the two most common ways to derive classes from IMGraphic:

- Geometry manipulation (also termed Simple IMGraphics). This mechanism defines objects that have specific geometry associated with them. The base IMGraphic protocol defines the ways you can draw and manipulate them. The supported subclasses correspond to the well defined 2D geometry set. For most 2D graphics geometry classes, for example, IGLine2D, a corresponding IMGraphic counterpart class, ILine, exists that supports object-specific needs such as rendering, hit detection, and so on. These subclasses act as wrapper classes for the geometry and thus have a 'has a' relationship with their geometry counterpart. You can also derive from IMGraphic to encapsulate more than one geometric primitive.

- IMGraphic manipulation (also termed Complex IMGraphics). This mechanism allows you to build a hierarchy of graphics to manipulate as a single graphic or group of graphics.

In addition to these two approaches, you can derive classes to create application specific subclasses, building application specific behavior into each of the subclasses. This is the most flexible approach. Use it to make classes more efficient for your specific applications.

NOTE: All concrete subclasses of IMGraphic must define, at minimum, the following pure virtual member functions:

```
virtual void draw(IGrafPort&) const = o;
virtual void transformBy(const IGrafMatrix&) = 0;
virtual IGRect2D geometricBounds() const = 0;
```

In addition to these mandatory member functions, subclasses should override some of the base implementations for performance gains, for example, translateBy, rotateBy, scaleBy, and so on.

IMGraphic Attribute Bundles

The properties that determine how a 2D graphic is drawn, such as fill color, frame color, and pen width, are called attributes. There are a number of classes for creating attribute bundles. Attribute bundles are instantiated and passed to a graphic to define the attributes for a particular graphic. If a graphic does not have an attribute bundle, the default graphic state of the root drawing port defines the graphic's attributes. See '2D Graphics Attributes and Bundles' for a detailed discussion of attributes.

The graphic bundle is heavyweight, so it is expensive to copy. Use adoptBundle() and orphanBundle() semantics when changing and modifying the IMGraphic bundle. These semantics limit applications to create objects on the heap only. Once adopted, the IMGraphic object takes full responsibility to properly destroy the bundle when the IMGraphic object is destroyed.

To modify an attribute of an IMGraphic object:

1. Orphan the bundle.
2. Change the attribute.
3. Have the IMGraphic adopt the bundle again.

This process keeps the IMGraphic base class simple and avoids duplicating IGrafBundle functions.

The IMGraphic base class provides default implementations for all bundle related member functions. Subclasses need not override this functionality, unless they have an attribute-based cache which needs to be invalidated or updated whenever the bundle is adopted and orphaned. For example, the result of looseFitBounds, when cached, needs to be invalidated (or reevaluated) when the attributes change. For performance reasons, subclasses should use lazy evaluation for rebuilding the caches.

IMGraphic has functions to adopt, browse, and orphan bundles.

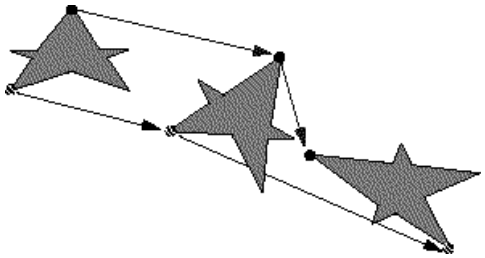
IMGraphic Transformation

Transformation is the process of altering every point in a graphic to resize (scale), reorient (rotate), or move (translate) the graphic.

Transformations can alter an IMGraphic's shape (by scaling and perspective transformation) and position (by rotating and moving). The transformation methods allow applications to change an existing IMGraphic's shape and location without having to recreate the IMGraphic.

IMGraphic transformation functions, translateBy, rotateBy, and scaleBy, provided for convenience and performance, let you transform a 2D graphic without using a matrix. You supply the values, and functions' default implementations create a temporary matrix and call the transformBy function, which takes an instance of IGrafMatrix as a parameter. Subclasses should override the default implementations to optimize for specific geometry and usage. Transformations change the points of the 2D geometry directly (except in the case of IEllipse) without copying the original points. IGrafMatrix is a 3-by-3 matrix for performing algebraic operations on a set of coordinate points (regular points, rational points, or vectors) that define a 2D graphic. IGrafMatrix is described at length in '2D Graphics Transformations.'

The figure displays a star undergoing various transformations:



Subclasses of IMGraphic can implement these transformations in two ways:

- IMGraphics that own an IGrafMatrix (such as IGraphicGroup) concatenate all transformations with this local matrix. The rendering pipeline uses the resulting concatenated matrix as the overriding matrix at the time of rendering the IMGraphic onto the specified IGrafPort. The local matrix is often referred to as the modeling matrix for the IMGraphic.
- Other IMGraphic subclasses, which are light weight and do not own a matrix, apply the transform directly to the geometry they own, thus actually changing the geometry. Transformation methods of all supported IMGraphic subclasses are operations without undo capability. If your application must implement an undo capability, save a copy of the IMGraphic prior to transforming it.

All IMGraphic subclasses are 'closed' to arbitrary transformations. An IGPolygon2D, when transformed by an arbitrary transformation, is still remains an IGPolygon2D. However certain 2D graphics geometries do not possess this closure property. For example, an IGRect2D when transformed by a perspective matrix is no longer a rectangle, and has no meaning of either width or height. The original specification of the rectangle (top left and the bottom right corner) are not sufficient to describe the transformed version of the rectangle. (IMGraphic does not support IRect for precisely this reason, however an IGRect2D can be represented by an IGPolygon2D.) All IMGraphic subclasses must be closed to arbitrary transformations.

All transformations applied are relative; you cannot 'untransform' a transformed IMGraphic by passing an identity matrix to the IMGraphic member function transformBy(). If your application must be capable of undoing the last transformation, transform the IMGraphic object by the inverse of the matrix used for the relative transformations. For example, undo transformBy(matrix) by the call transformBy (matrix.invert()) or rotateBy(30.0) by rotateBy(-30.0). The matrix must be invertible and round-off errors can occur.

IMGraphic Bounds

All 2D graphics classes have bounds. The bounds of a graphic are defined by the smallest IGRect2D that encloses the geometry of the graphic.

The IMGraphic inquiry member functions allow you to inquire about certain specific properties of the graphics object without modifying the object itself. Any graphical representation of a geometry has at least two different types of bounds:

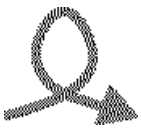
- Loose fit bounds
- Geometric bounds

The most common use of the inquiry functions is to detect whether a graphic is selected (hit or picked). The geometricBounds and looseFitBounds functions are fast and suitable when speed is more important than accuracy.

The 2D Graphics framework uses both the supported bounds inquiry member functions frequently, so the functions must be fast and efficient. Your concrete subclasses should cache these bounds if computing them is expensive.

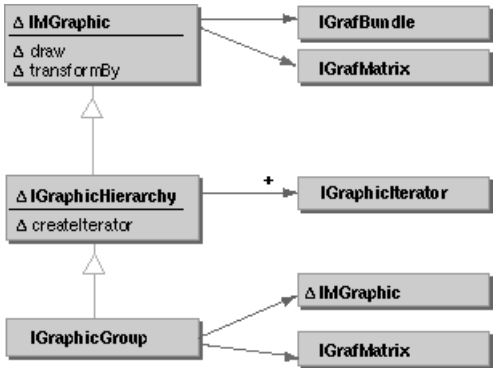
These methods do not follow the strict guidelines of bit wise consts and are labeled as abstract const methods. Because they are abstract consts, they are unsafe when stored either in shared memory or in ROM (specially used for servers).

This figure displays the IMGraphic used in discussions of bounds mechanisms:



IGraphic Hierarchies and Groups

Class architecture shows that IGraphicGroup derives from IGraphicHierarchy, which derives from IMGraphic.



IGraphicGroup provides a concrete implementation of iterating its children. The IGraphicIterator created iterates only one level. If you need to iterate more than one level deep, create iterators on subsequent IGraphicGroups.

IGraphicHierarchy

IGraphicHierarchy is an abstract base class that defines a hierarchy protocol for graphics that are grouped together into a graphic conglomerate.

You can create groups or hierarchies to descend from the base class IGraphicHierarchy for making available the iterator polymorphically.

The 2D Graphics framework provides a concrete subclass of IGraphicHierarchy, IGraphicGroup, which allows creation of trees of graphics to allow manipulation of a conglomerate group.

IGraphicGroup

IGraphicGroup is a concrete class, derived from IGraphicHierarchy, that implements a group of graphics as a tree structure. It allows you to combine 2D graphics into a unit that is treated as a whole. IGraphicGroup inherits directly from IMGraphic and thus each of the nodes owns its own IGrafBundle and can thus affect its own side of the hierarchy.

IGraphicGroup, as an IMGraphic subclass, references other IMGraphic objects. Although all manipulative behavior of complex IMGraphics is similar to simple IMGraphics, unlike the previous types, these objects do not completely encapsulate IMGraphic objects they refer to. IGraphicGroup descends from the abstract base class IGraphicHierarchy which makes available polymorphically the methods to create iterators for traversing groups.

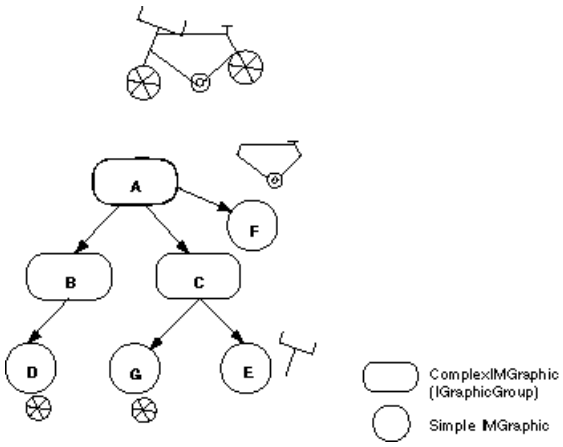
IGraphicGroup does not provide support for DAGs (Directed Acyclic Graphics).

IGraphicGroup does not allow its children to have more than one parent in a team. (IGraphicGroup does no checks for this you must check for multiple references).

The destructor of IGraphicGroup destroys itself and all its children.

IGraphicGroup Relationships

IGraphicGroup creates a collection of IMGraphic objects forming a group. Each of the IMGraphic objects can be an IGraphicGroup itself, so one can create a hierarchy of objects. The figure shows an example of a hierarchy created by IGraphicGroup to create the image of a simplified bicycle:



In the figure, the elements have the following relationships:

- A, B and C are IGraphicGroups.
- D, E, F and G are different simple IMGraphics encapsulating more than one geometry.
- A has references to the following elements:
  - B, the rear wheel
  - C, the front wheel
  - F, the body of the bike
- B refers to D.
- C refers to the IMGraphics G and E.

Groups B and C control the transformations associated to the rear and the front wheel respectively. The two wheel are represented by the primitive geometries D and G. E represents the handle-bar of the bike. Thus moving node C moves both the front wheel and the handle-bar. Moving node A moves the entire bike.

While applying transformation matrix to the children at the time of rendering, the group creates a temporary IGrafPort object and concatenates its matrix with that stored in the IGrafPort. This new IGrafPort renders its children and is destroyed once the child is completely rendered. The IGrafPort objects are created on the stack.

IGraphicGroup Attribute and Transformation Hierarchy

Each IGraphicGroup, if it so chooses, defines its own attributes and transformation. By default the attribute bundle is NIL and the transformation matrix is set to identity. As IGraphicGroup is a complex IMGraphic, it has references to other IMGraphics, its children. By definition, each of the child must inherit the attribute traits and transformations of its parent. However, since each child can be multiply referenced, it inherits these by concatenating the parents information, without modifying its own at the time of rendering. The concatenation of these attributes is achieved at the time of the draw call.

Both the attribute and the matrix are concatenated with the IGrafPort object which is passed as a parameter to the draw call.

The IGraphicGroup constructors let you instantiate a group with an attribute bundle. There is a default constructor.

IGraphicGroup(IGrafBundle* adoptedBundle);

In addition, each graphic adopted into a group can have its own attribute bundle. The attribute bundle of the graphic overrides the attribute bundle for the group into which the graphic is adopted.

In the bicycle example described in IGraphicGroup relationships, attributes and transformations of object A (body of bike) are concatenated with the IGrafPort object passed to A (as a parameter to the member function draw) and a new IGrafPort object, APortObject, is created on the stack. APortObject is passed to object C which concatenates its state and creates a new port object, CPortObject. This new CPortObject is passed to object E to be rendered. Object E concatenates its state with CPortObject and renders itself using this new state.

Matrix concatenation is simple post concatenation of the parent's matrix with that of the child. Thus, the matrix used to render object E is:

(E's Matrix) * (C's Matrix) * (A's Matrix)

In the wagon example, described in Complex IMGraphic, the wagon group has no attribute bundle so it has a default bundle. When the wagon body is adopted into the wagon group, it has a bundle that specifies fill and frame attributes that override the default bundle of the wagon group. When the wagon wheels are adopted into the wagon group, they also have attribute bundles that specify fill and frame attributes that override the default bundle of the wagon group. Because the wagon body and wagon wheel groups are at the same level in the hierarchy, their bundles have no affect on each other.

To transform the entire group, apply a transformation to the top-most graphic in the hierarchy. To transform only a part of the entire group, apply a transformation to the appropriate nested group. You can, of course, always transform any graphic before you adopt it into its group.

In the wagon example, a translation is applied to the wagon group to make the wagon body and four wheels move a specified distance. A rotation is applied to each wheel group so that the wheels rotate in addition to the translation.

The transformation matrix of the parent is post-concatenated with the matrix of the child. For example, the transformation and rotation matrix for the wagon wheels is found by multiplying the wagon group transformation matrix by the wagon wheel rotation matrix as follows:

((wagon wheel rotation matrix) * (wagon group translation matrix))

IGraphic Hierarchy and Group Iteration

Since IGraphicGroup allows creation of hierarchies, support for iterating the hierarchy has been built in this base class to be available polymorphically.

IGraphicIterator

This function is pure virtual in the abstract base class IGraphicHierarchy and all subclasses must provide an implementation. Subclasses who do not wish to expose their children may return an empty iterator when this member function is invoked.

IGraphicIterator* createGraphicIterator() const = 0;

This function creates an IGraphicIterator which iterates through the first level of a hierarchy. In the bicycle example, the iterator created for IGraphicGroup, A iterates over B, C and F. To iterate further, you must create iterators for both B and C. All subclasses creating hierarchies must provide a concrete implementation.

IGraphicIterator is an active iterator that facilitates the iteration over the children of an IGraphicHierarchy. IGraphicIterator functions include:

const IMGraphic *IGraphicIterator::first();
const IMGraphic *IGraphicIterator::next();
const IMGraphic *IGraphicIterator::last();

The createIterator function is called by the IGraphicGroup::draw function. The returned iterator covers one level of the group. The entire group is covered because the IGraphicGroup::draw functions are called recursively.

IGraphicIterator IGraphicHierarchy::createIterator();

IGraphicIterator has first, next, and last functions for traversing the group. The order that graphics are placed into the group is important because it affects the drawing order. This order is determined by the adoptFirst and adoptLast functions of the IGraphicGroup.

2D graphics are added to a group with the adoptFirst and adoptLast functions. Function adoptFirst puts the adopted graphic into the first position in the group; adoptLast puts the adopted graphic into the last position in the group. You nest groups by adopting a IGraphicGroup instance, which is derived from IMGraphic. There is no limit to how deep you can nest groups, but the same graphic cannot be adopted by more than one group.

IGraphicGroup::adoptFirst(MGraphic* graphic);
IGraphicGroup::adoptLast(MGraphic* graphic);

IGraphicGroupIterator

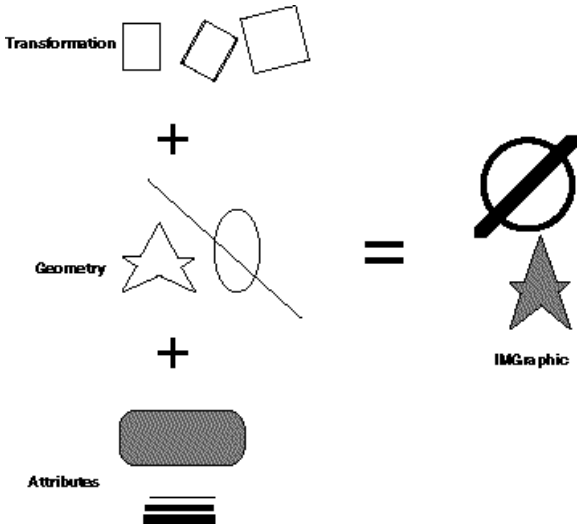
IGraphicGroupIterator provides functions to traverse and manage a hierarchical tree of graphic objects.

Modeling Examples

This section provides examples of IMGraphic subclasses and conglomerate groups of graphics.

IMGraphic Basic Components

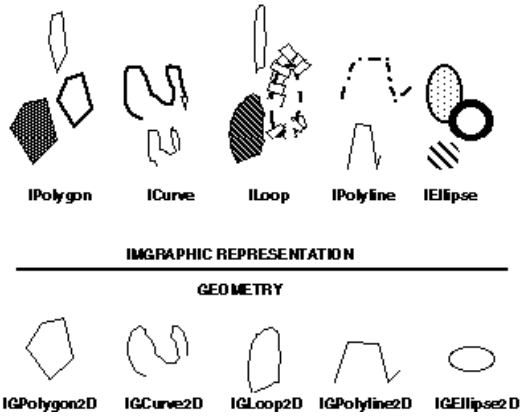
The figure shows the basic components of IMGraphic:



IMGraphic and Corresponding Geometries

If your application needs to hold geometry-related data along with its graphical representation, use IMGraphic and its subclasses. You can also implement your own retained mode implementations similar to IMGraphic or extend IMGraphic for their particular application.

The figure shows various IMGraphics and their corresponding geometries.



The 2D Graphics framework supports other concrete IMGraphic subclasses:

- IGraphicGroup
- IImage
- IArea

User-defined IMGraphic Subclasses

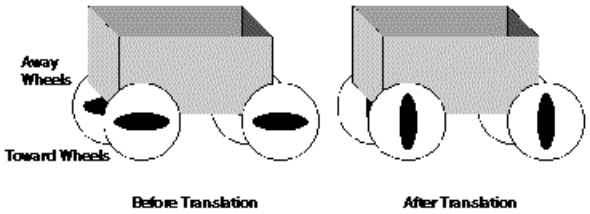
The figure shows simple and complex user-defined subclasses of IMGraphic.

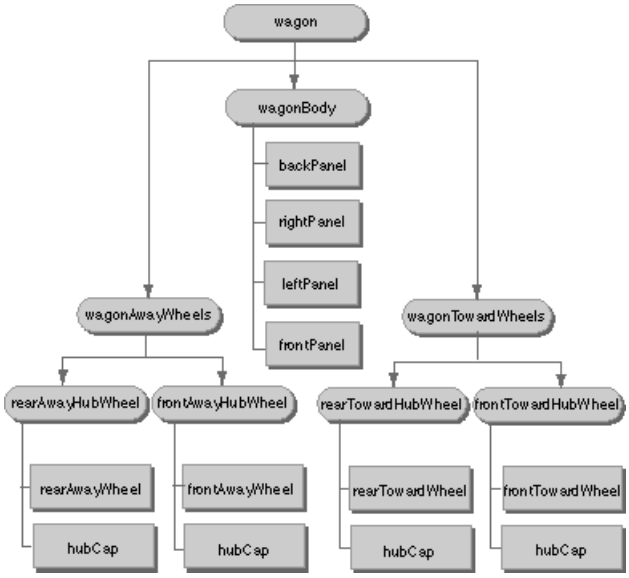


Complex IMGraphic

A complex 2D graphic consists of a number of interrelated shapes. A wagon has a body, four wheels, and a handle. When the wagon moves, the wheels rotate. A 2D graphic group lets you create a complex 2D graphic such as a wagon that you can hit test, transform, and draw as a hierarchical unit.

In the wagon shown in the figure, the body consists of four panels and rests on four wheels. When the wagon moves, the wheels move too, but also rotate.





Chapter 20. 2-D Graphics Rendering Pipeline

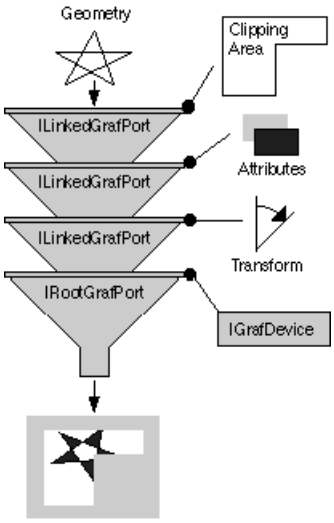
2D Graphics Rendering Pipeline Overview

The rendering pipeline model for the 2D Graphics framework is the IGrafPort class. Through this class, you can associate graphics attribute states, transformation matrix, and clipping areas as the state for a given port. You can render geometry by using IGrafPort with the state applied along the rendering pipeline and the states will remain stored in the port.

At the root IGrafPort pipeline or chain, an IGrafDevice deals with all the aspects of rendering geometry with respect to the state set.

The IGrafPort is the component for constructing a hierarchy of graphics states. The IGrafPort can be constructed by specifying a link to the parent IGrafPort and some state information. The state information does not have to be the complete information required to render the geometry, because the chain of IGrafPorts eventually provides enough information to render the geometry. In the event of missing state information, the mechanism inherits the values from its parent. IRootGrafPort has the default values for all attributes.

The figure provides an overview of the rendering pipeline:



The 2D Graphics framework provides two distinct ways of rendering geometries on a device.

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry through a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

Immediate and retained mode: In pure immediate mode rendering, the application objects are rendered without keeping any record of the information (geometry,

© Copyright IBM Corp. 1998

attributes, hierarchies) used to render them, while in retained mode, the record of all objects, their hierarchies, and other related information is stored for rendering.

GrafPorts

The IGrafPort class is a convenient container of the graphic state that is associated with the undefined native rendering device, such as the GDI interface for Windows. Its most important function is that it supplies a set of nonvirtual draw calls that make it convenient for drawing geometries.

For convenience, the IGrafPort class provides 2D draw calls.

The large number of disjoint calls (as opposed to a polymorphic single call) enforces the rule that only a small, well-defined set of basic 2D geometries is supported by graphics devices.

IGrafPort also provides 2D draw calls that take an additional bundle and model matrix for convenience in certain specialized conditions. The draw calls use the graphics state set by subclasses of IGrafPort. ILinkedGrafPort is a very efficient way to set up hierarchical graphics states. You can make these states shared among the appropriate geometry objects to be extended. The graphic state includes attribute state (bundle), clipping area, and transformation matrices.

There are two kinds of transform matrices in rendering:

- The view matrix describes the view coordinate space. Every geometry and every attribute associated with geometry is subject to transform when rendered.
- The model matrix is the matrix describes the model coordinate space. Only the geometry is subject to transform, not the attributes.

Member functions allow you to access the graphic state.

The 2D Graphics framework provides two distinct ways of rendering geometries on a device.

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry through a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

Note: Immediate and retained mode: In pure immediate mode rendering, the application objects are rendered without keeping any record of the information (geometry, attributes, hierarchies) used to render them, while in retained mode, the record of all objects, their hierarchies, and other related information is stored for rendering.

IGrafPort subclasses

A subclass of IGrafPort inherits the set of convenient draw functions in the base class.

IBaseRootGrafPort

The IBaseRootGrafPort class contains an IRootGrafState object, which contains default values for every possible substate that is needed for rendering a geometry. It has a default graphic state setting.

IBaseRootGrafPort uses world_coordinate units to draw all the graphics. One unit is equivalent to one point (1/72 of an inch). Graphics that draw to IBaseRootGrafPort are resolution independent.

ILinkedGrafPort

An ILinkedGrafPort can associate itself with one of the three types of graphic state information. These three types of graphic state are attributes, transformation matrix, and clipping information. The attributes are represented by an IAttributeState object. IAttributeState is an abstract base class of IGrafBundle, IFillBundle, and so on. The transformation matrix is represented by an IGrafMatrix object. The clipping information is represented by an IGArea object.

With this class, one can construct a hierarchy of 'linked' ports each representing a different state. All state information are kept hierarchically so it is easy for one to change some states without reconstructing the hierarchy.

IStatefulGrafPort (special class)

The IStatefulGrafPort class is similar to IBaseRootGrafPort because it contains an IRootGrafState object, which has a full set of graphic states that is needed for rendering a geometry. It also has a pointer to an IGrafDevice object. The difference from the IBaseRootGrafPort is that you can set the value of the graphic states individually, and therefore it provides greater flexibility. This gives you a simpler API, but you have the responsibility of determining which of the states needs to be set and structuring the graphic state information. This is not as efficient as ILinkedGrafPort because the way hierarchical state linking can be shared among draw callers.

Note: This class is provided for experimental use and is subject to change. Use it for special case implementation only.

For any primitives that the native platform does not support, a subclass of IGrafDevice called IFrameBuffer can perform appropriate rendering in software to form the raster image. Along the way of rendering, the order of complexity of a high-level geometry with the specific attributes, transform, and clipping states can be reduced to a simpler geometry successively, until either a native operation can be carried out for such a geometry, or the resulting image can be constructed at the end. For example, a circle defined by a center and radius is to be rendered with a gray pattern fill. If the native rendering engine does not support gray fill pattern for conics but does for polygons, the circle is discretized to a polygon and the native polygon with gray fill pattern is performed.

GrafPort Subclass Notes

It may be that there will be a case that an image is created by loading from an images file and is drawn to both IBaseRootGrafPort and IRootGrafPort. The size of the image displayed in IBaseRootGrafPort will be bigger than the size of the image

in IRootGrafPort. But if the image is used as a fill pattern to fill graphics in both IBaseRootGrafPort and IRootGrafPort, the images in both GrafPorts will be the same size.

Special GrafPorts in Open Class

By default the current coordinate system for the 2D Graphics framework is left-handed. In order to support dual coordinate systems and multiple mapping modes inherently for existing OpenClass clients, IExtendedRootGrafPort subclasses or extends IBaseRootGrafPort.

For traditional Cartesian coordinate system (used in OS/2 GPI), use the kRightHand parameter.

```
class IExtendedRootGrafPortPort : public IBaseRootGrafPort {
public:
IExtendedRootGrafPortPort(
IPresSpaceHandle adoptedDeviceContext,
EDeviceCoordinateSystem = kLeftHand,
EDeviceMappingMode = kPixel);
};
```

The enum type EDeviceMappingMode defines the modes of device mapping: kPixel, kLowMetric, kHighMetric, kLowEnglish, kHighEnglish, kTwips.

Also for convenience, a subclass of IExtendedRootGrafPort, IRootGrafPort is provided. The coordinate system in IRootGrafPort is upperLeft only.

By default, IExtendedRootGrafPort and IRootGrafPort are pixel-based. All graphics on screen, except text, are drawn in pixels when an IExtendedRootGrafPort or IRootGrafPort is used. When using either of these GrafPorts, text is always drawn in world_coordinate units.

Chapter 21. 2-D Graphics Transformations

2D Graphic Transformations Overview

IGrafMatrix implements the full set of 2D linear transformations including pseudo perspective. Most users of graphic transforms use IGrafMatrix as ready made classes. In addition to implementing transformations, IGrafMatrix implements a number of matrix utilities including concatenation, inversion, computing the determinant and transposing the matrix. IGrafMatrix provides the following:

- Matrix Transformations: identity, translation, scaling, rotation, affine, and perspective.
- Matrix Elements: concatenation, mathematical operations, perspective mapping.

You can also transform graphics using the IMGraphic transformation functions described in '2D Graphics Framework.'

The 2D Graphics framework uses transformations to transform points, rational points, and vectors. A vector has magnitude and direction but not location. Vectors can only be scaled, rotated, or sheared from the origin. Points and rational points have location, and they can be translated. They have an extra homogeneous coordinate 'w'. The extra coordinate allows affine transformations such as translation, and rotation around an arbitrary point. Points have an implied w == 1.0.

IGrafMatrix provides the following features:

- Batch transformations.
- Queries, to see if the pipeline can take advantage of fast rendering algorithms.
- Accelerators to speed up typical rendering transformation.
- Caching.

IGrafMatrix is not a general math package. It does not solve sets of linear equations or eigenvalue problems, and it does not have arbitrary dimension. It is designed for the kinds of linear transformations (where the coordinate space is modified linearly) needed by graphics, geometries, and the rendering pipeline.

Useful Information

For general discussions of points, rational points, and vectors, see 'Mathematical Elements for Computer Graphics,' section 2-15 'Translation and Homogeneous Coordinates.'

For information on perspective mapping, see Paul Heckbert's UC Berkeley Masters Thesis: 'Fundamentals of Texture Mapping and Image Warping,' Appendix A.2 'Inferring Projective Mappings from 2-D Quadrilaterals.'

The Transformation Matrix

IGrafMatrix defines a 2D transformation matrix. A 2D transformation matrix is an array of numbers with three rows and three columns for performing algebraic operations on a set of homogeneous coordinate points (regular points, rational points, or vectors) that define a 2D graphic.

© Copyright IBM Corp. 1998

IGrafMatrix transforms points in homogeneous space using rational points represented by the IGRPoint class. In homogeneous space, the third IGRPoint coordinate (fW) allows translation, rotation around an arbitrary point, scaling around an arbitrary point, perspective transformations, and shear transformations in addition to rotating and scaling around the origin through matrix multiplication. IGPoints have an implied fW of 1.0. Scaling a homogeneous matrix by a scalar does not change how it transforms an IGPoint; however, scaling a homogenous matrix by a scalar does change how it transforms an IGRPoint because an IGRPoint exists in homogeneous space.

IGrafMatrix implements the full set of 2-D linear affine transformations (translate, rotate, and scale) and supports concatenation (multiplying one matrix by another matrix). You can derive from IGrafMatrix to define your own transformation matrix implementation. The following table lists the IGrafMatrix member functions. All functions have default implementations that you do not need to override. Refer to the online Class and Member Function descriptions for information on the IGrafMatrix functions.

Constructors

IGrafMatrix functions can be called directly. The default constructor creates an identity matrix. The other constructors create rotate, translate, and scale matrices. In most cases, you can avoid having to specify or know about the numbers in the matrix. One of the constructors takes a specified array of numbers; and the matrix, element, and setElement functions provide access to the matrix.

IGrafMatrix Inquiry Functions

The following table describes the IGrafMatrix inquiry functions:

Member function	Action
matrix	Fills in a 9-element array of GCoordinates.
rotate	Returns true iff (if and only if) the matrix is a rotation matrix. If the matrix is a rotation matrix, rotate fills in the angle between 0 and 360 degrees, and the center of rotation.
type	Returns if the matrix is identity, translate, scale around a point, rotate around a point, affine, or perspective.
isIdentity	Returns true iff the matrix is identity.
isTranslate	Returns true iff the matrix only translates points.
isAffine	Returns true iff perspectiveX==0, perspectiveY==0 and homogeneous==1. A rectangle passed through an affine matrix becomes a parallelogram.
isRectilinear	Returns true iff a rectangle passed through the matrix is still a rectangle.

IGrafMatrix Mathematical Operations

The following table describes the IGrafMatrix mathematical operations:

Member function	Action
invert	Allows a matrix to reverse the effect it has on points. For example, if a transform moves a point from (0,0) to (1,1), the inverse transformation will move a point from (1,1) back to (0,0). If the transform cannot be inverted, throws an exception and does not change the destination.
untransformPoint	Is a shortcut method to invert on the fly
determinant	Gets the determinant. If the determinant is 0, a matrix is not invertible.
makeAdjoint	Converts the matrix to its 'Classical Adjoint.'
transpose	Swaps the rows and columns of a matrix.
normalize	Divides the matrix by the homogeneous value. Normalize raises an eUndefinedTransform ITransformException if the homogeneous value is 0.0.

Matrix Transformations

IGrafMatrix has a set of calls to support the common graphics transformations: identity, translate, scale and rotate. An empty constructor creates an identity matrix. These calls allow you to:

- Construct translate, scale and rotate matrices.
- Apply a translate, scale, or rotate to the matrix (relative transformation). Relative transformations are shortcuts for concatenating the matrix with a translate, scale or rotate matrix.
- Set a matrix to one of these transformations identity, translate, scale and rotate (absolute transformation).

To set up a translate, rotate, or scaling matrix, call the translateBy, rotateBy, and scaleBy functions. To set up a matrix that will, for example, first scale and then rotate the graphic, call the scaleBy and rotateBy functions in sequence. The matrix stores the cumulative transformation so you can apply one matrix to various graphics instead of scaling and rotating each graphic explicitly.

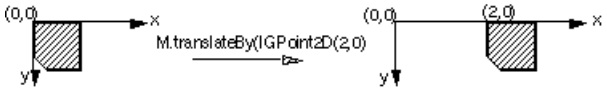
The setToRotate, setToScale, and setToTranslate functions set the matrix to rotate, scale, or translate, respectively. These functions cannot be called in sequence like rotateBy, scaleBy, and translateBy. If you call them in sequence, only the last call has any effect.

Identity

An identity transformation does nothing when applied to a set of points and vectors. It sets the destination to the source when you call transformPoint(s) or unTransformPoint.

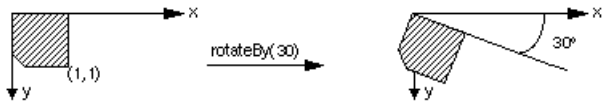
Translation

For translation, a delta point specifies the amount an object moves along the x and y axis. Function translateBy(IGPoint2D(2,0)) moves an object from the origin along the x axis by 2 units.



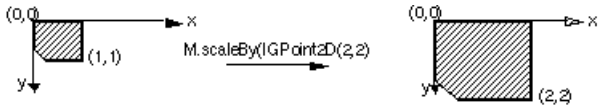
Rotation

The angle of rotation is in degrees and is measured from the positive x axis to the positive y axis. Rotation occurs around a point. The default rotation is about the origin.

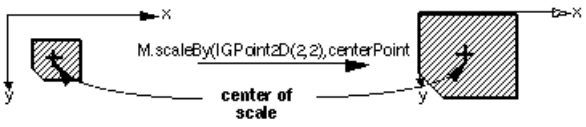


Scaling

A point specifies scaling factors for x and y. A scale factor with x==y is a uniform scale. If a scale factor is greater than one, it enlarges the object. If the value is between zero and one, it shrinks the object. A scale factor that is less than zero mirrors an object.



Scaling is relative to the center of scale. That is, a point at the center doesn't change. Change increases the further a point is from the center. The default center of scale is the origin: (0,0).



Matrix Concatenation

Matrices can be concatenated with other matrices. Concatenation is equivalent to sticking two matrices together such that a point first passes through one matrix and then the next during a transformation. The functions translateBy, scaleBy, and rotateBy are just concatenation shortcuts for translate, scale, and rotate matrices.

Think of concatenation as putting a transformation into a 'list' of transformations. Later when transformPoint is called, the point is passed through each transform in order. For linear matrices, the 'list' is compressed into a single matrix.

Concatenation is not commutative, that is if A and B are matrices, then A times B is not necessarily equal to B times A. For example, rotation matrix times a scale matrix does not necessarily give the same result as a scale matrix times a rotation matrix even though the component matrices are identical. This means it matters where the new transformation is placed in the 'list'. For this reason the 2D Graphics framework defines the notion of pre-multiplication and post-multiplication. Every function has a corresponding pre-multiply method with a prefix 'pre' in front of it. The function that does not have the prefix has an implicit meaning of post-multiplication. The preRotateBy, preScaleBy, and preTranslateBy functions premultiply the specified transformation matrix by whatever values exist in the parameter list.

To create a sequence of arbitrary transformations rather than just translation, scaling, and rotation, use the concatWith function. Function concatWith concatenates two transformation matrices so that during the transformation each point passes first through one matrix and then the next. Function concatWith places the specified transformation matrix after the existing matrix.

Note: For speed, the two concatenated matrices are formed into one matrix internally.

You can use the preConcatWith function to place the specified transformation matrix before the existing matrix.

Perspective Map Transformations

You can define a transformation matrix by specifying what a convex quadrilateral (four-sided polygon) looks like before and after the transformation. This technique utilizes a perspective map. A perspective map defines a linear transformation from a source convex quadrilateral to a destination quadrilateral.

You can:

- Use the IGrafMatrix::setToPerspectiveMap function to create a perspective map to define all of the standard transformations
- Define a translation matrix by specifying a source polygon and a destination polygon that is the same as the source, but in a different (x, y) position
- Define a scaling matrix by making the destination polygon larger or smaller than the source
- Define a rotation matrix by rotating the destination polygon.

Function setToPerspectiveMap sets the matrix to a perspective mapping from a source polygon to a destination polygon. The source and destination polygons must be convex, and they must have four vertices (quadrilateral). It creates the following perspective mapping:

fromQuadrilateral [0] —> toQuadrilateral [0]
fromQuadrilateral [1] —> toQuadrilateral [1]
fromQuadrilateral [2] —> toQuadrilateral [2]
fromQuadrilateral [3] —> toQuadrilateral [3]

You can use a perspective map to define all of the standard transformations:

- Moving the quadrilateral defines translation.
- Making the quadrilateral larger or smaller defines scale.
- Rotating the quadrilateral defines rotation.

You can describe affine and perspective transformations. You can even easily describe shear transformations.

Matrix Elements

A linear matrix is defined by the coefficients in a set of linear equations. For the two-dimensional case the matrix looks like this:

```
           | scaleX     shearY     perspectiveX |
(x,y,w) *  | shearX     scaleY     perspectiveY |  = (x',y',w')
           | translateX translateY homogeneous  |
```

Where:

```
 x' = x*scaleX + y*shearX + w*translateX
 y' = x*shearY + y*scaleY + w*translateY
 w' = x*perspectiveX + y*perspectiveY + w*homogeneous
```

Note: Matrix Elements are intended for advanced users only.

Chapter 22. Multimedia Classes

Multimedia Devices

WIN OS/2

A multimedia application that integrates text and graphics with a combination of audio, motion video, images, and animation makes your application more attractive to the user, easier to use, and offers better mapping to real world objects.

You can use the window control classes and multimedia classes to implement an interface for your application that looks like the controls of common electronic devices, such as stereo components and video cassette recorders (VCRs). Your application can use these controls as interfaces to control audio and video media that is presented to the user.

Keep in mind user expectations when you create your user interface. Mapping your user interface to the mental image the user has of real-world devices greatly enhances the ease of use of your product. For instance, most users are familiar with the play, stop, pause, fast forward, and reverse controls of an audio cassette recorder.

To use the multimedia classes, ensure that your working environment meets the following requirements:

Software

Windows Sound System or IBM OS/2 Version 3.0 Multimedia Presentation Manager/2 (MMPM/2) Version 1.1 is required for using the Open Class Library multimedia classes.

Hardware

If you are:	Hardware required:
Using MIDI, audio classes, or software motion video sound	Windows or OS/2-supported sound card
Using CDXA or AudioCD	CDROM (OS/2 only)
Hearing any sounds (CD or otherwise)	Speakers or headphones
Recording audio	Microphone

A *medium* is a carrier of information. A *multimedia computer system* is one that is capable of input or output on more than one medium. With the new class of computers, information in virtually any format can be combined into multimedia presentations.

Multiple types of input allow the user to interact with the computer in a style that best suits the information to be communicated, thus relieving overloaded input channels, such as a keyboard, mouse, or microphone.

Output information can be presented in a variety of formats. Typically, output implies a computer display, video, or audio. Video has the potential to hold people's interest and illustrate concepts better than static images. Audio and speech contribute a unique quality to the multimedia system and can increase the information's content.

© Copyright IBM Corp. 1998

Multimedia Device Classes

The Open Class Library supports audio adapters, CD-ROM drives, video-disc players, logical devices, amplifier-mixers, and other hardware devices as media devices. These media devices are abstracted into classes that contain the data and functions essential for the operation of the real-world devices that they model.

The classes you define for your application combine the capabilities of several classes. Before defining the objects your application needs, choose real-world models that the user knows how to manipulate for the interfaces. You can then use the appropriate Open Class Library multimedia classes that provide the corresponding functions.

The base device classes let you create multimedia devices for your application. The following table lists the base classes and refers to the appropriate sections describing the multimedia devices. In addition, these sections describe how to use the devices.

You can directly instantiate device objects from the following classes:

Device	Class
Audio amplifier-mixer	IMMAmpMixer
CD audio player	IMMAudioCD
CD Extended-Architecture player	IMMCDXA
Digital video player	IMMDigitalVideo
Master audio	IMMMasterAudio
MIDI sequencer	IMMSequencer
Waveform audio player	IMMWaveAudio

Abstract Device Classes

Common functions for devices are made available through *abstract device classes*. That is, abstract device classes allow inheriting classes to reuse common functions. Note that you cannot instantiate objects from these classes. The following sections describe the multimedia base class and abstract device classes. The multimedia base class (IMMDevice) is the parent class for the family of multimedia classes, including the base device classes and the other abstract classes.

Device
Purpose

IMMPlayableDevice
Used for many tasks, such as playing, pausing, and seeking.

IMMFileMedia
Used for devices that work with files.

IMMRecordable
Records, saves, cuts, pastes, allows undo, allows redo, saves-as.

IMMRemovableMedia
Opens and ejects media; unlocks and locks doors.

Base Functions for Multimedia

All of the multimedia device classes inherit from the IMMDevice class. This abstract class contains all of the common functions for device objects. These functions include the following:

- Querying the capabilities of a device
- Opening and closing devices
- Changing the speed and time formats
- Changing the audio (on or off)
- Controlling the volume

Playable Device

Objects are usually instantiated from this class in applications that manage different types of devices (such as a VCR and CD remote). You do not create an actual device rather the instantiated object is used to point to a device a user wants to activate (such as a video player).

An object instantiated from IMMPlayableDevice is capable of performing tasks that a home device does to play such media as CDs or video tapes. In addition to the common device functions, such as play, pause, and seek for devices that support playback, IMMPlayableDevice objects are able to perform resume, stop, and query position and length functions.

WIN When using the multimedia classes on the Windows operating system, for audio and video devices you need to load the appropriate file before using any functions on those devices. Otherwise, an exception is thrown.

Audio Devices

This section introduces creating audio devices that play wave and musical instrument digital interface (MIDI) file formats.

Audio input and output is usually in the form of wave or MIDI files.

There is a distinction between sound and music; while it might not be distinctive to a radio, it is to a computer. Sound, such as the sound in wave files a Waveform Audio Player Device is basically just digitized data that a computer cannot process. MIDI augments waveform audio by producing sound in the multimedia environment. Your system plays whatever is in a wave file out to your speakers. By comparison, music is actual information. The following section introduces basic concepts about these two formats.

MIDI is a standardized set of data blocks or 'messages' that instructs any MIDI-compatible sound source as to which notes to play. Rather than representing actual sound recordings, as a file of digitized audio does, a MIDI file merely describes what notes to play and includes settings for the sound or instrument, duration, stereo pan position (how far left or right), and volume.

A MIDI file is comprised of variable-length chunks. There are two types of chunks: a header chunk and one or more track chunks. The number of chunks are defined in the header. A MIDI event can be one of a number of things. It can be a message that turns a particular note on or off, that changes the voice being played by a particular channel, or that defines something about the piece being played.

When you create multimedia applications that play instrumental music, handle it with MIDI (music) rather than digitized audio files (sound). The relative size of the files involved is one of the best arguments for doing so. Compact discs (CDs) are common and hold large amounts of information.

Digital synthesis methods, either FM or wavetable playback, are customarily driven by MIDI.

MIDI files have the extension .mid and deliver more music per byte than other formats. MIDI files are comparable in size to ASCII text files, while the other music and sound formats (for example, wave) are comparable to color bitmaps. A digital audio recording of a musical instrument performance can consume large amounts of storage; a MIDI file describing that same performance can take only a kilobyte (K) or two. Wave and CD audio files can sometimes be too big to distribute easily, whereas MIDI files are smaller. Compare 5 minutes of sound in a MIDI file to a 20 MB-wave file. The MIDI file takes about 10K of storage whereas the wave file takes 20 MB. There is a noticeable difference with your application's performance.

MIDI files do not support voice or words. The main role for MIDI in multimedia is music composition and production. Once the music is recorded, it can be played on a high-end synthesizer and recorded in wave or CD-audio formats.

Waveform Concepts

Waveform refers to a digital representation of an original audio sound wave. *Audio* refers to sound waves that have a perceived effect on the human ear.

Digital recordings offer more consistency than MIDI files. A CD recording of music sounds virtually the same on any CD player you use, but a MIDI musical file could sound like, for instance, a French horn on one synthesizer and a kazoo on another. The sound depends on the quality of the sound card. However, MIDI music typically sounds cleaner, more realistic, and more professional than the digital recording, especially if you do not have a sound studio to record your tracks.

Wave files have the extension .wav and contain analog sound that has been recorded digitally. The pieces of sound are usually sampled sound stored as data. An *analog-to-digital converter* creates sampled sound. A wave file can reproduce sound with anything from telephone to compact disc quality in monaural or stereo under computer control.

Chapter 23. Application Control

Multi-processing

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:

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
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.

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 (Windows NT, OS/2, and AIX) 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 methods that

© Copyright IBM Corp. 1998

are themselves protected from simultaneous access to multiple threads of execution are explicitly documented as 'Thread-Safe.' Assume any Open Class class or method that is not documented as Thread-Safe to not be Thread-Safe.

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.

Data Sharing

The behavior of a multi-threaded process behavior is non-deterministic; you don't know:

- When a thread may enter a method.
- How many threads may be entering the same method concurrently.
- How many threads may be concurrently entering methods that access the same data.

When C++ data is only accessible through methods, synchronizing the access to those methods 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:

- 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 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 Method

The application object's method does not know which thread is calling on it, but can use the ICurrentThread class to obtain a reference to the IThread or INonGUIThread object: use IThread::current. Open Class allows only a single object of ICurrentThread. You can use ICurrentThread to

- call application methods 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

To restrict the threaded execution of an application method to the thread currently running, override ICurrentThread::run to call the application method.

Locking a Resource

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*() {

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

Event-driven Synchronization

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 .

Unshared Thread 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.

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/rest its own priority within the process. Both process priorities and thread priorities are specified as enums.

Process scheduling uses the following enum:

enum INonGUIApplication::EProcessPriority

Thread scheduling uses the following enum:

enum INonGUIThread::EThreadPriority

New or Enhanced Process/Thread Classes

The following table lists the Open Class classes that are new or extended in this release:

Class	Functionality
IExternalProcess	Basic facilities for starting and controlling the execution of processes
IThreadLocalStorage	Provides a portable way to create and access per-thread global data
IEnvironment	Specifies the execution environment for use with a process
ICondition	A class to be used for thread synchronization. When used in conjunction with IResource, provides the classic 'Monitors and Conditions' construct
IPrivateCondition	Concrete subclass of ICondition, for use within a process
ISharedCondition	Concrete subclass of ICondition, for use between processes on a single computer

Counted Pointers

ICountedPointerTo is a templatized 'smart pointer' class which mimics a C++ pointer and can be used for reference counting both IMRefCounted objects and objects that do *not* descend from IMRefCounted.

ICountedPointerTo 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 chapter Using Counted Pointers.

Reference Counted Objects

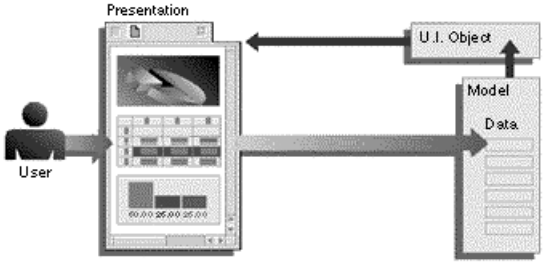
IMRefCounted is a public base class that can be used by any class that needs reference-counting semantics, similar to the IRefCounted class provided by the IBM Open Class library in the past. This version of IMRefCounted differs in two ways:

- IMRefCounted is *thread safe*. That is, its `addRef`, `removeRef`, and `count` methods 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 smart pointer classes such as ICountedPointerTo.

Chapter 24. Notification Classes

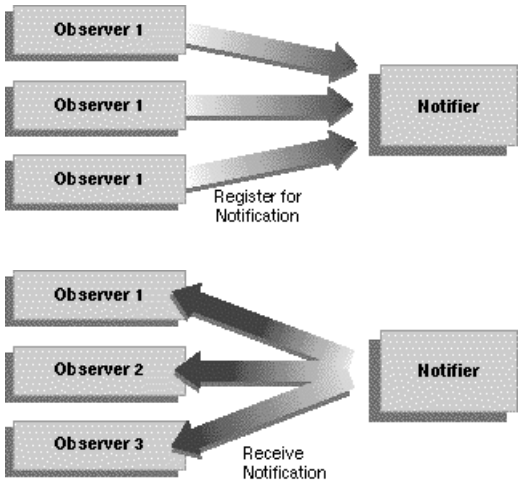
Event Notification

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.



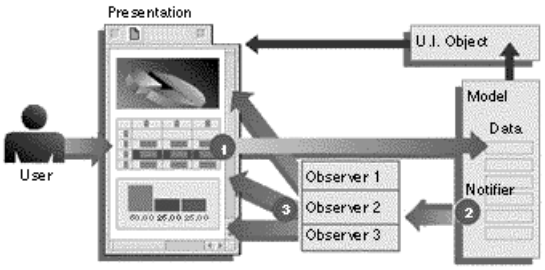
Your model should *notify* the view when the model's data is modified so that the view can redraw itself to reflect the change.

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. Once registered, when that object is modified, it issues notification to the registered observer objects, passing a *notification event*.



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

© Copyright IBM Corp. 1998



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.

This version of IBM Open Class extends its previous 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. The Notification classes can be used with nonframework components (regular IBM Open Class applications) as well as with framework components (Component-based applications). They can be used for either synchronous or asynchronous notification, and for either single-threaded or multithreaded applications.

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.

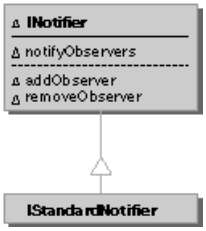
- *Interests* are used by observers to identify the specific event for which they wish notification.
- *Notification Events* identify the notification type and are passed from the notifier to one or more observers.

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.

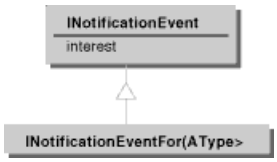
You can streamline your notification mechanisms by filtering the types of notification an observer receives. The notifier contains a set of notification IDs that defines all the possible types of changes, and observers can register to be notified only of the types of changes they care about. The following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

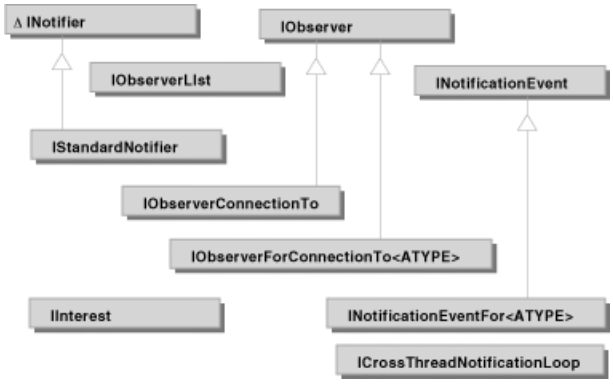
INotificationEvent encapsulates a single notification event. Notification event objects identify the notification type, and are passed from the notifier to the observers.



INotificationEventFor is a templatized class used for passing event data from a notifier to its observers.

The Notification Family of Classes

The following is a view of all the notification classes:



Implement a Notification

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. INotify::notifyObservers() provides a synchronous implementation and INotify: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 INotify::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();
```

Chapter 25. Error Handling and Testing - Exceptions

The Exceptions Mechanism

Exceptions provide a mechanism for dealing with unusual circumstances in programs. The exception classes have the following characteristics:

- A stack of exception message text strings. These strings allow you to describe the exception in detail.
- An error ID that lets you uniquely identify what error caused the exception.
- A severity code that lets you determine whether the exception can be recovered from or not.
- Information about where the exception was thrown.

The exception classes' member functions allow you to:

- Add information about where the exception was thrown.
- Add text to the description of the exception.
- Get the error ID of the exception.
- Determine if the exception is recoverable.
- Log the exception data.
- Set the error ID of the exception.
- Set the severity of the exception.
- Set a trace function.

Using Exceptions

There are three primary ways to use the exception classes:

1. Certain functions in IBM class libraries throw exceptions that are derived from the IException class. If you are familiar with the characteristics of the exception classes, you can take advantage of this knowledge to make code that uses the IBM class libraries more robust.
2. You can both throw and catch exceptions in your own code. The exception classes provide a convenient way to package information about an exception.
3. You can derive your own exception classes.

© Copyright IBM Corp. 1998

Chapter 26. Application Testing

Overview of Application Testing

The VisualAge C++ Test Framework helps you create and run your test cases. It provides base classes with a standard format for writing tests, accepting inputs, and reporting results. You just need to write a test subclass and override certain functions. The Test Framework also provides, through runtest macros, a main() function that controls the execution of tests. You include these macros to let the Test Framework run your tests and deal with the flow of control.

Use the Test Framework to perform the following tasks:

- Ensure that a target test class runs properly
- Group tests, so that more than one test is performed at a time
- Perform the timing measurements of a test class
- Test an entire tree of classes that adheres to the same protocol

Test Framework Components

The main components of the Test Framework are:

- Test classes
- Runtest macros

Test Classes

A *target class* is a class in your program that you want to test for correct performance. You write a *test class*, derived from a Test Framework class, that runs the target class and does the evaluation.

ITest is the abstract base class from which all test classes are derived, including the test class that you write and declare to be a friend of your target class. You can derive directly from ITest to perform simple tests. In addition, three kinds of functionality are provided by specialized subclasses derived from ITest:

- To group tests, so that you can perform more than one test at a time, use the multiple tests: ITestCollection and ITestMultiplexer.
- To perform timing measurements, use the timing tests: ITimingTest and IStartStopTimingTest.
- To test an entire tree of classes adhering to the same protocol, use the Protocol tests: IComparisonTestOf and IStreamTestOf.

The test classes work by creating and evaluating an instance of your target class. You do not have to make changes to your target class code to test its behavior.

Runtest macros

The Test Framework defines several macros that you may use to create applications to run your tests:

- To automatically generate a main function to run a single test, use runTestImplementationMacro.

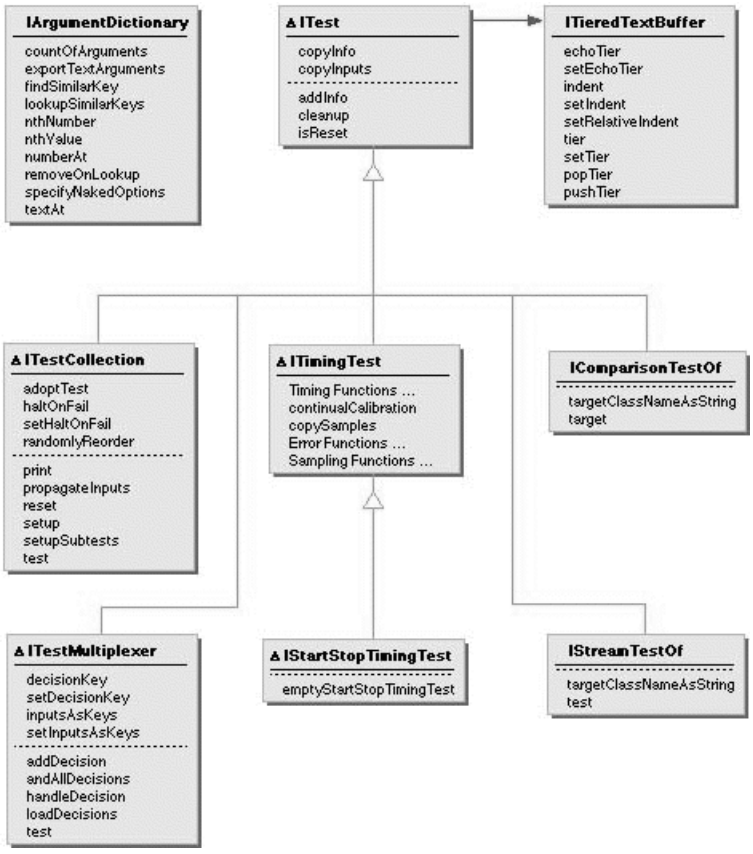
© Copyright IBM Corp. 1998

- To create your own application to run several tests, use beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro.
- For protocol tests, use comparisonTestMacro and IStreamTestMacro.

Test Framework Architecture

The core of the Test Framework consists of the base class ITest and a number of associated classes, which can be subdivided into several categories according to their characteristics:

- Multiple tests: ITestCollection, ITestMultiplexer
- Timing tests: ITimingTest, IStartStopTimingTest
- Protocol tests: IComparisonTestOf, IStreamTestOf
- Auxiliary classes: ITieredTextBuffer, IArgumentDictionary
- Test macros: runTestImplementationMacro, beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, runTestResultMacro, comparisonTestMacro, IStreamTestMacro



The ITest Base Class

The core of the Test Framework is the class ITest, an abstract base class that defines the protocol for an executable test. ITest is a framework that embodies the concepts of:

- A test success state (not set, pass, or fail)
- A target object on which tests are performed

- Input through a collection of IString objects
- Output via an object with ostream-like behavior
- Textual metainformation about the test
- Uniform exception handling by the Test Framework

To write a test, you generally write a subclass of ITest. This subclass overrides the test function and, optionally, the associated framework functions reset, setup, and cleanup. To run the test, you generally use the runTestImplementationMacro to create an application that defines the main function to create and run the test.

The Decision Function

The Decision Function Overview

You can group your tests by combining multiple operations into a single test class.

A *decision function* is the code you write that determines if the code you are testing does what you expect. Using a single ITest object to test several functions allows the test decision functions to share code and functions.

The ITestCollection Class

ITestCollection is a subclass of ITest that tests a collection of ITest objects. This allows you to group related ITest subclasses together within a single test. The tests in the collection are run sequentially to determine the success or failure of the entire group. The resulting test passes if all of its subtests pass and fails otherwise. By default, the subtests are always executed in a fixed order (the order in which you add subtests). The subtests can be run in random order if you call the randomlyReorder function after adding subtests. By default, ITestCollection doesn't halt on subtest failure. It runs all the subtests by default. To change this default behavior, you need to call setHaltOnFail(true) in your setupSubtests.

The ITestMultiplexer Class

ITestMultiplexer is a subclass of ITest that supports the model of multiple decision functions applied to a single test target. These decision functions are invoked using test keys. You write a group of decision functions, then build a table that maps keys to functions. This helps you avoid the duplication of setup and cleanup code for the functions that share them.

Keys are case-sensitive IString objects that select the decision function to run when the test function is called. They may be used in three ways:

- A single key may be specified via setDecisionKey.
- The key kAllDecisions may be specified via setDecisionKey, with the result that all decision keys are called.
- A certain number of ITest inputs may be interpreted as keys, and those decision functions run. You must call setInputsAsKeys with the number of inputs to be used as keys.

The default behavior is to run all decisions.

Timing Tests

ITimingTest

The ITimingTest base class enables you to write tests that measure the time taken by a specific operation to complete. The derived class IStartStopTimingTest allows you more control in timing events.

ITimingTest is the principle class used to measure CPU performance. It measures the time taken to execute a well-defined operation—for example, adding a member to an IACollection subclass.

ITimingTest expects the operation to show some variation in execution time because of multitasking, varying CPU load, and other variable conditions. Consequently, the class takes multiple measurements and reports the median.

ITimingTest contains three framework functions: TimingSetup, TimingTest, and TimingCleanup. Only the TimingTest function is timed. The TimingSetup and TimingCleanup functions run before and after TimingTest, but are not themselves timed.

IStartStopTimingTest

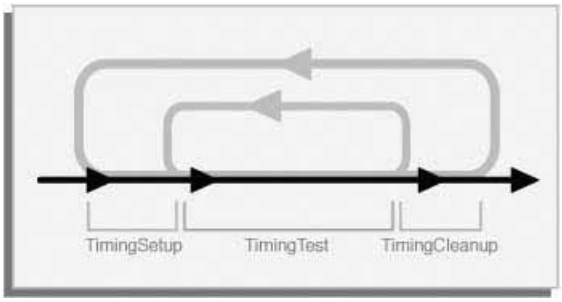
When possible, you should use ITimingTest because it has been designed to have very high accuracy. In some cases, however, you will not be able to use ITimingTest. For example, consider a class INetThing that queries the network. Only the first instantiation of INetThing performs a certain action. If you wished to test the initial object-constructor time of INetThing, you would need to delete each existing INetThing before the next one is constructed. What you need is a sequence like the following within the timing loop:

1. Begin timing.
2. Instantiate INetThing.
3. Stop timing.
4. Delete INetThing.

IStartStopTimingTest makes this kind of timing test possible at the cost of a loss of precision. Because it takes a substantial amount of time to actually measure time, when the timer is started or stopped, IStartStopTimingTest can only measure times in milliseconds.

Structure of Loops

The time needed to execute the timingTest function is measured by taking a number of samples—the *sample count*. Each sample consists of a call to timingSetup, a number of calls to timingTest, and a call to timingCleanup.



The calls to timingTest happen in a tight loop—the *timing loop*. Subclasses or the caller can control how many iterations comprise the timing loop by changing the *timing count*. The total time taken by the timing loop is divided by the number of iterations to get a single *sample time*. The median of all sample times is taken as the estimate of the time to execute the TimingTest function.

Accuracy

Calls to the timingTest function incur an overhead because of the time required to make the virtual function call. ITimingTest compensates for this by computing an empty function time, which is the estimated time to make a call to an empty virtual function. This time is subtracted from each sample time in order to calibrate for this overhead.

Another way to increase the accuracy of timing tests is to increase the timing count to divide the various sources of error across more iterations. In general, the value of the timing count should be at least 100.

Tolerance

ITimingTest usually runs in *automatic mode*. In this mode, samples are taken sequentially until the error for the estimated time declines to some percentage. This percentage is the *tolerance*. For example, if the tolerance is 0.05, then samples will continue to be taken until the time measurement has an error of plus or minus 5%.

Automatic sampling estimates the mean of a moving window of samples. This window is *sample count* samples wide. The error of the mean is half of the 95% confidence interval. When this error (expressed as a fraction) declines below the tolerance, then the window is accepted and its mean returned as the estimated time.

It is possible that the time being measured might never stabilize, and that the required tolerance might never be reached. To handle this contingency, an upper limit has been set on the number of samples taken during automatic sampling. If the maximum number of samples windows is taken, and samples still have not converged, a warning is issued and the final window is accepted.

You might not want to use automatic timing for some tests. If automatic timing is disabled, then the number of samples specified by sample count is taken and their median returned as the measured time, regardless of the error. In this case, tolerance is ignored.

Success and Failure

A timing test fails:

- If some function calls `setSuccess(false)`
- If the time measurement itself is considered to fail
- If it results in a value less than zero, which indicates a problem with calibration

In the absence of these events, if the measurement reaches the required tolerance within the specified maximum allowed time, the test succeeds.

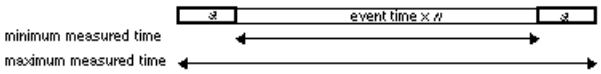
Precision

The timing test class attempts to correctly report the precision of its results by applying the following mechanisms:

- The standard deviation of all samples is taken. This is used to compute a 95% confidence interval for the average.
- Timing overhead is measured by calling the internal timer repeatedly and measuring the time required to do so.

The contribution of timing overhead to errors in event-time measurement is estimated as follows:

The bar represents a single sample. The two boxes labeled *a* at the ends of the bar represent the timing overhead. The left end of the box is the time at which the call to a timing function is initiated, and the right end of the box is the time at which the call returns. The resultant value may represent time at any point within the box. The minimum measured time is exactly the event time multiplied by the number of iterations. The maximum measured time is the event time multiplied by the number of iterations plus 2*a*. This is the worst case. The actual event time falls somewhere in between.



ITimingTest reports the most pessimistic interpretation reasonable. It reports each timing measurement as the upper end of the 95% confidence interval for the event time, plus an upper-bound estimate of 2*a/n* for the timing overhead. A greater sample count decreases error by reducing the size of the confidence interval, a larger timing count decreases error by distributing the timing overhead over more events.

ITimingTest parses command line arguments. These arguments modify the way in which timing is done. They may also be supplied programmatically (see ITest).

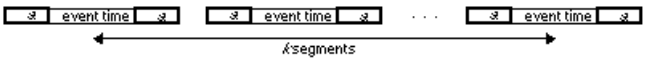
Argument	Description
-at tol	Set the tolerance for automatic sampling to tol (default = 0.1).
-c count	Set the operations per sample to count (default = 100).
-cc	Continually calibrate by measuring empty loop time before each sample.

-co correlation	Set the minimum correlation between samples for which to post an error to correlation (default = 0.4).
-m maxTime	Consider this test to fail if the median time per operation is greater than maxTime (default = none).
-na	Turn off automatic timing.
-s n	Take n samples, each sample consisting of many operations (default = 10).
-t	Don't do timing tests; just execute one operation without timing.
-w maxWindows	Stop if times do not converge within maxWindows windows of automatic sampling (default = 10).

The start-stop timing test attempts to correctly report the precision of its results by estimating the dispersion of samples and the sampling error as follows:

- Each sample consists of one or more segments. These segments represent timed events, beginning with a call to start an internal timer, the activity being timed, and a call to stop the internal timer. Let *k* be the number of segments per sample.
- The Test Framework makes the pessimistic assumption that the measured time for each segment has a potential error range of 2*a*, where *a* is the timing overhead.

The contribution of timing overhead to errors in IStartStopTimingTest event-time measurement is also estimated. The boxes labeled *a* at the ends of the bar represent the timing overhead. This diagram represents a single sample, which consists of one or more segments. Each segment is bracketed on each end by the timing overhead.



- The minimum measured time is exactly the event time multiplied by the number of segments, *k*. The maximum measured time is the event time times *k* plus 2*ka*. Thus, the worst-case error is 2*ka*.
- As is the case with ITimingTest, increasing the sample count reduces the dispersion between samples. Unlike ITimingTest, however, increasing the timing count does nothing to reduce the sampling error because the timing overhead is incurred repeatedly within the sampling loop, rather than only at the start and finish.

Protocol Tests

The Test Framework contains two ready-made protocol tests implemented using template classes:

- IComparisonTestOf
- IStreamTestOf

Protocol tests help leverage testing effort by allowing you to test classes when those classes are expected to adhere to some protocol.

If your object uses the default comparator and default streamer, protocol macros let you write entire protocol tests with a single line.

You can use the macros whenever you can create the target with a single parameter list. You cannot use the macros if you need to create or reference other objects in order to create your target. In this case, you must derive your protocol test class from the appropriate template class.

Auxiliary Classes

The Test Framework defines two auxiliary classes: ITieredTextBuffer and IArgumentDictionary.

ITieredTextBuffer

ITieredTextBuffer behaves like the C++ ostream class. It contains << operators for all basic types. It can filter output so that detailed information is suppressed or displayed. The text is printed directly to the console (and log file, if the log option is specified). Each instance of itest contains a ITieredTextBuffer to which subclasses may stream diagnostic text messages. ITest itself uses this mechanism to report progress and results.

IArgumentDictionary

IArgumentDictionary is a general-purpose class for parsing text arguments on a command line. It takes as input an ordered collection of IString objects and parses them as arguments on a command line into pairs of keys and values. This allows you to quickly check for the existence of a keyword on the command line or to retrieve the value given for a certain option.

A leading-hyphen character identifies keywords. Anything without a leading hyphen is a value argument. A keyword picks up the following argument if it is not another keyword.

This example shows how IArgumentDictionary parses the command-line input to a test to create the key-value pairs:

```
-parm -n sample1 Joan Tom -ccc 84 85
```

The following table explains the options used in the example.

Key	Value	Description
-parm		The value associated with -parm is an empty IText, not NIL. This allows you to distinguish 'There is no -parm keyword' from 'There is a -parm argument with no associated value.'
-n	sample1	The -n argument picks up the following argument, 'sample1,' as its value.

1	Joan	There is no associated keyword, so 'Joan' is assigned key 1. The key is an IText object, not a numeric value.
2	Tom	There is no associated keyword, so 'Tom' is assigned key 2. The key is an IText object, not a numeric value.
-ccc	84	The -ccc argument picks up the following argument, '84,' as its value.
3	85	There is no associated keyword, so '85' is assigned key 3. The key is an IText object, not a numeric value.

You also may specify, with the specifyNakedOptions function, that certain keywords never take value arguments. Such keywords are called naked options. Here is another example:

```
-s 20 -l -o sampout
```

The following table explains the options used in the example.

Key	Value	Description
-s	20	The -s argument picks up the following argument, '20,' as its value. The value must be between 1 and 256.
-l		The -l represens the loging option, which takes no arguments. If a value follows the -l, the value is taken to be a keyless option and is assigned the key 1. The key is an IText object, not a numeric value.
-o	sampout	The -o represents the output-file option, which takes a single argument.

Test Macros

The Test Framework defines macros that you may use to create applications to run your tests.

The runTestImplementationMacro defines an application to run a single test. The beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro macros may be used within a main routine to create an application that can run several tests.

Applications created using these macros recognize several options: -t, -e, -n, and -o options (see below). The -test option must be specified when running the application.

Command-line options understood by the application created using the macros are shown in the following table:

-e[cho] h|g|n|d|D	Set detail of diagnostic output: headline, general, normal, detail, Debug
-n numberOfRuns	Run the test numberOfRuns times; default is to run test once.
-o[ptions]	Pass further arguments (after the -o) to the ITest subclass.
-t[est] class	Run the ITest subclass class.
-log [logfilename]	Write the output into logfilename.

runTestImplementationMacro

The runTestImplementationMacro macro expands to define the entire main routine to create and run a single test. It takes the name of the test class for its parameter. The test object is created and the run function is called for the test. The main routine returns with a 0 indicating that the test succeeded; otherwise, 1 is returned.

beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, runTestResultMacro

Use beginTestFrameworkMacro, runTestMacro, endTestFrameworkMacro, and runTestResultMacro to create an application that can run various tests instead of just one. The following table summarizes each of these macros:

beginTestFrameworkMacro	Takes argc and argv for its parameters. This macro expands the code to parse the command-line arguments for options.
runTestMacro	Takes the name of the test class for its parameter. This macro expands the code to create and run the text.
endTestFrameworkMacro	Takes no parameters. This macro expands the code to check if a test was run and prints out an error if no test was run.
runTestResultMacro	Returns a boolean indicating whether test succeeded.

To use these macros you must write your own main routine:

1. In main routine, use beginTestFrameworkMacro with the argc/argv values.
2. Use runTestMacro for each test class you wish to run.
3. Use the endTestFrameworkMacro and runTestResultMacro to get the result of the test.

Only the test specified with the -t option will be run when the application executes.



Printed in the USA