Content Manager

Software Version 24.3

COM SDK

opentext™

Document Release Date: August 2019
Software Release Date: July 2024

Legal notices

Documentation updates

The title page of this document contains the following identifying information:

- Software Version number, which indicates the software version.
- Document Release Date, which changes each time the document is updated.
- Software Release Date, which indicates the release date of this version of the software.

To check for updated documentation, visit https://www.microfocus.com/support-and-services/documentation/.

Support

Visit the MySupport portal to access contact information and details about the products, services, and support that OpenText offers.

This portal also provides customer self-solve capabilities. It gives you a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the MySupport portal to:

- View information about all services that Support offers
- Submit and track service requests
- Contact customer support
- Search for knowledge documents of interest
- View software vulnerability alerts
- Enter into discussions with other software customers
- Download software patches
- Manage software licenses, downloads, and support contracts

Many areas of the portal require you to sign in. If you need an account, you can create one when prompted to sign in.

Contents

Content Manager COM SDK

This document describes the Component Object Model Software Development Kit (COM SDK) for Content Manager. It provides an introduction to the design and content of the COM SDK, it gives instructions and guidance for using the various tools and objects, and is the logical starting point for the COM SDK documentation suite.

For those wanting to understand the capabilities of the COM SDK, this document can be read on its own. For those intending to use the COM SDK, it serves as an orientation and introduction. For a complete technical understanding, you can add a reference to the COM SDK (TRIMSDK.DLL) into the object browser of your chosen Integrated Development Environment (IDE), where you will be able to access detailed helpstrings for each object, method and property within the COM SDK.

Effective integration of Content Manager with other applications using the COM SDK requires a technical understanding of it's tools, as well as a user perspective of the Content Manager application in general and (most importantly) a business understanding of the particular implementation of Content Manager and any other application for which an integration is required.

Using the Content Manager COM SDK

Technical Prerequisites and Assumptions

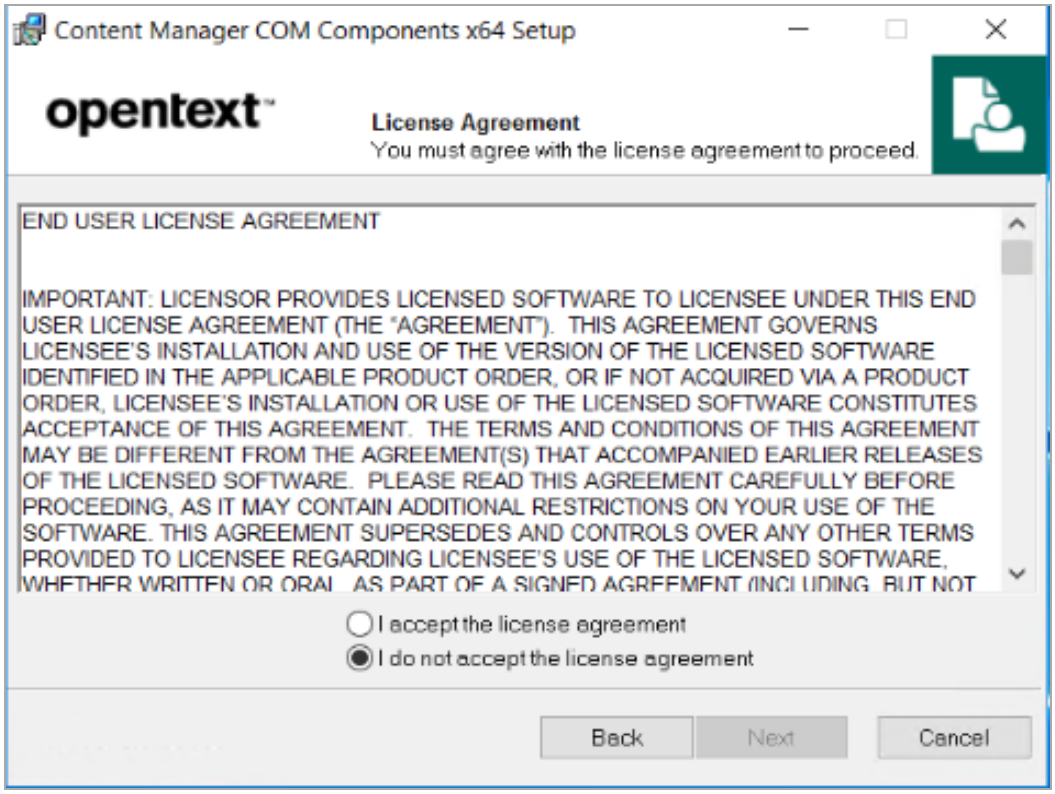
As the name implies, the Content Manager COM SDK components are based on Microsoft's Component Object Model (COM) standard. This documentation assumes the reader is a programmer with an understanding of COM programming principles, the structure of COM-compliant object models (i.e. objects, interfaces, methods and properties) and some experience of using a COM-compliant programming language such as Visual Basic or C++. The code examples in the documentation are in Visual Basic as well as C#, although any COM-compliant language can be used. The Visual Basic examples are tested using the Visual Basic 6 compiler, while the C# examples are tested using Microsoft Visual Studio .NET 2013, requiring the .NET Framework Version 4.5.

Using Content Manager COM SDK with .NET Applications

The Primary Interop Assembly (TRIMPIA20.DLL) for the COM SDK (TRIMSDK.DLL) is no longer shipped with Content Manager. It is possible for developers wishing to access this SDK using .NET languages to add a reference to the COM SDK and create their own Secondary Interop Assembly, although they are encouraged to use the native .NET SDK (HP.HPTRIM.SDK.DLL) instead.

For programmers who wish to write a .NET application using the COM SDK, they need to run the **CM_COMComponents_x86.msi** or **CM_COMComponents_x64.msi** file to register the COM SDK.

1. With elevated user rights, from the installation media, run **CM_COMComponents_x86.msi** or **CM_COMComponents_x64.msi** file. The Welcome dialog appears.
2. Click **Next**. The License Agreement dialog appears.



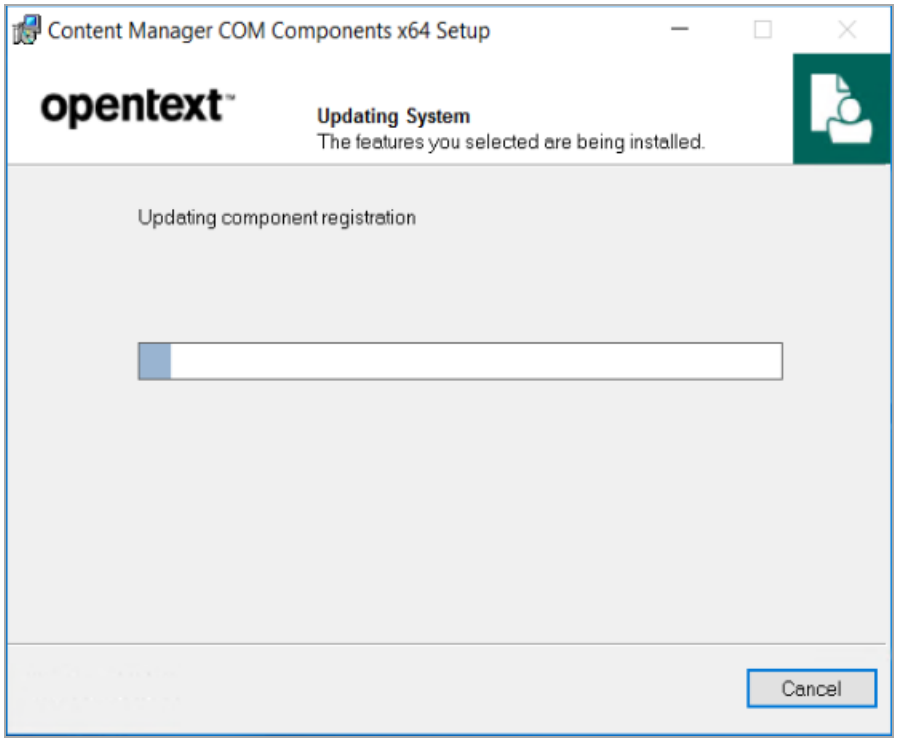
3. Select **I accept the license agreement** and click **Next**.

   The Destination Folder dialog appears.

4. Click **Next** to keep the default installation folder (recommended).

   The Ready to Install the Application dialog appears.

5. Click **Next**. The Updating System dialog appears.



The setup status dialog is displayed.

6. Click **Finish** to exit the setup wizard.

A Short History of the COM SDK

In the early days of the Content Manager (previously known as TRIM Records & Document Management – from creation up to version 4.1 and Records Manager), it was generally not possible to programatically access the TRIM Database without resorting to writing SQL. This was no easy task. The programmer had to fully understand the data schema and the complex data relationships to translate a user's requirement into a set of SQL statements. It was even more daunting if any data was to be modified, as it was necessary for the programmer to understand the application business rules to avoid corrupting the Database.

To help clients avoid these pitfalls, when version 4.2 of TRIM was released in 1997, it included an extra program that gave programmatic access to some of the internal application functionality through an small subset of functions labelled the TRIM API. By using this API instead of SQL to access the TRIM data, the business rules were automatically applied and therefore the Database integrity was preserved. The TRIM 4.3 release expanded the capabilities of the API, exposing more of TRIM's underlying functionality and thereby enabling a wide variety of integration possibilities with other applications. The business opportunities that this created has ensured that this version of the API has been widely used as a successful means of seamlessly integrating TRIM data and functionality into other applications. The TRIM 4.3 API was an out-of-process COM Automation server, implemented in a file called "tsapi.exe".

In the version 5.0 release of TRIM, the application has been given a major design overhaul, creating a more robust, scalable enterprise architecture and a brand new look-and-feel. As this was a "from-the-ground-up" redevelopment, the API was replaced by a comprehensive library of functions known as the Software Development Kit. For the first time, most of the TRIM business functions can be

accessed programatically. This gives the programmer vastly more control over the application and far greater scope for integration than was previously possible.

Since version 6.0 of TRIM, new in-process server (HP.HPTRIM.SDK.DLL) has been added for .NET solutions. This is covered in separate document.

TRIM and Records Manager are now referred to as Content Manager.

What is the Content Manager COM SDK?

The Content Manager COM SDK is a suite of tools that allow programmers to create custom solutions, services and integrated applications by leveraging the functionality of Content Manager using COM. These tools give Content Manager clients and third-party integrators the opportunity to ERDM-enable line-of-business applications, to create custom document-centric applications, and to increase the return-on-investment of an organisation's information assets, such as Classification systems, controlled vocabularies, and knowledge repositories.

The COM SDK is an in-process server implemented in a dynamic link library (DLL) file. This method of implementation means that the COM SDK is loaded into the same memory space as the application that invokes it. The result is fast execution of methods, and it also enables a number of separate applications to work with different Databases through the same component.

At the core of the COM SDK is a comprehensive model of all business objects in Content Manager. All data fields associated with these objects are exposed as properties through the COM SDK, and various methods are provided to implement common application functionality. With very few exceptions, it is possible to programmatically access via the COM SDK all aspects of Content Manager that are available through the client user interface.

The security model is common to both the COM SDK and the Content Manager client application – therefore any custom process that calls the COM SDK must connect to the Database using the current user's login, and this user's Content Manager security profile determines the extent of data and functionality that the process can access.

Better Building Blocks

Content Manager has been designed in such a way that all the business objects used in the application are automatically exposed. This means that using the Content Manager COM SDK is very efficient, and all the same underlying objects and properties that the application's programmers use to build the Content Manager user interface are also available to third-party programmers via the COM SDK.

Hear, Say

As an Automation server, the Content Manager COM SDK defines the methods and properties that an Automation client (a program or macro) can invoke. Another way of thinking of this is that the COM SDK is the set of questions that Content Manager can answer, and the commands that it will obey.

The client and server relationship is sometimes called the master/slave relationship: the former does all the talking, and the latter does all the doing (and does not speak until spoken to).

The COM SDK also includes a special class called TRIMEventProcessor. This is an outbound interface, meaning that Content Manager calls methods on this interface in response to events that occur within Content Manager. The integration programmer can write code that implements these methods, and therefore can execute code in a server process in response to actions in the Content Manager client.

ActiveX Controls (TSJOCX.DLL)

The Content Manager client installation includes a number of ActiveX controls contained in a module called TSJOCX.DLL. Whilst this module continues to ship with Content Manager, the use of the

controls within it will be on an “at your own risk” basis and OpenText no longer provides ongoing support and maintenance for these controls.

The Content Manager Object Model

Objects and Interfaces

To understand how objects in the Content Manager COM SDK are used, you must have a basic understanding of objects and interfaces, and how they are used in the Component Object Model, otherwise known as COM. Interfaces are the key to understanding COM, and directly affect the way COM objects in the SDK (and other COM-based object models) are used in code.

An object is an instance of a class, where a class is some common type of entity in an application. Typical classes of objects in the Content Manager application are Records, Record Types and Locations. Each class of object has properties, which represent the named data attributes that are present on each instance of the class.

Generic Interfaces

In the Content Manager COM SDK, there are certain common interfaces that are implemented by many different objects, in addition to the object's own default interface. The benefit of this is that different types of objects can be treated polymorphically, using the common interface. It also allows extensibility of the COMSDK, such that new types of objects can be introduced in later versions, without breaking the existing interface.

The common interfaces in the Content Manager COM SDK are:

Interface	Implemented By
IBaseObject	Any objects that can be created and deleted independently.
IBaseChildObject	Any object that can only exist as a 'child' of a Base object.
IBaseObjects	Any collection of Base objects.
IBaseChildObjects	Any collection of Base Child objects.

Which common interface is implemented depends on the type of object, i.e. whether it is considered a Base Object or a Child Object (or a collection of either of the two).

Base Objects

A Base Object is any first-order entity in the Content Manager Database. It exists independently of other objects (although it may be related to many objects) and can be explicitly created or deleted by a user with the appropriate authority.

Examples of base objects are Records, Locations, Record Types, Keywords, Schedules, Document Stores, and many others.

Base Objects are also called 'persistent' objects because they persist after the program code that manipulates them is not executing (that is, the data for the object can be saved in the Content Manager Database and retrieved later to recreate the object).

All Base Objects have an internal Unique Row Identifier (URI) that uniquely identifies different objects of the same type, and most have a corresponding unique name or other identifier (such as Record

Number) that is visible to the user. Whenever a persistent object is modified in the SDK, you must call a 'Save' method on the object to commit the changes to the Content Manager Database.

Child Objects

Child objects, on the other hand, only exist as a dependent of another object.

Examples of Child objects are Requests (children of a Record), Addresses (children of a Location), and Lookup Items (children of a Lookup Set).

Generally speaking, child objects are created indirectly, as a result of performing some task on a base object or a dependent collection object.

Some child objects represent a relationship between Base objects, such as Record Locations, Record Keywords, and Related Records. Adding or removing child objects does not directly affect the 'parent' Base objects (for example, the AttachedKeyword object represents a Keyword (from the Thesaurus) associated with a particular Record). Each instance of this object defines a relationship between a Record object and a Keyword object, but if the AttachedKeyword instance is removed, the Record and the Keyword objects themselves are unaffected.

Because of the dependence upon the parent object, child objects cannot be independently saved, but the data they contain is persisted when the parent object is saved.

Collections

Collections are a special class of objects that are used to temporarily hold and manipulate a set of several objects of the same type. They have a standard interface that allows the programmer to access the items in the collection directly by index position or to iterate through the collection sequentially. The standard convention in the SDK is that a collection object takes the plural name of the type of object that it contains (for example, the Locations collection is used to hold multiple Location objects).

If a given object implements the IBaseObject interface, then the Collection of those objects will implement the IBaseObjects interface. Similarly, if a given object implements the IBaseChildObject interface, then the Collection of those objects will implement the IBaseChildObjects interface.

Collections for Base and Child Objects

Dependent or Child objects are those that can be manipulated like other objects in the SDK but which cannot be independently created or saved. These are usually dependent upon one or more persistent objects in the Database, and often represent relationships rather than tangible objects. Typically, Child objects are held in collections that are accessed via a property of their parent object.

An example of a dependent collection is the AttachedKeywords collection. This can be used like any other collection object to navigate to the keywords it contains. However, even though the collection can be modified (by adding or removing relationships between keyword terms and the record), it is not independently saved. The information that the child collection represents is saved when the object on which it is dependent is saved. In the case of the AttachedKeywords example, the relationship between the record and the AttachedKeyword is saved in the Record object.

Methods and Properties

In the previous section we discussed the relationship between objects in the Object Model, and how objects implement predefined interfaces. Each interface is defined as a specific set of methods and

properties, and it is through these that the object provides its functionality and data.

The definition of each method and property is provided in the reference section of this documentation. However, most object classes can be used according to generic processes, and these are discussed in the next sections on the Object Model.

Common Properties

Many objects implement Properties with the same names. This makes it easier for the programmer to learn the object model. Although not all objects implement these properties, the meaning is consistent for those that do.

Property	
Database	Returns the Database object that created this object.
Uri	Returns the internal number that uniquely identifies this object.
Verified	Returns True if the current state of the object's data is valid.
Type	Returns the object type for this object (for example, Record, Location etc).
ErrorMessage	Returns the description of the last error associated with this object.

Common Methods

As with common Properties, many objects implement Methods with the same (or similar) names.

Method	Description
GetProperty	Returns the data value of a property identified by a property Id.
SetProperty	Sets the data value of a property identified by a property Id.
GetPropertiesAsString	Returns the data values of a set of properties for the object, as a string formatted for the use specified.
Verify	Checks the validity of the current state of the object's data.
Save	Saves the current state of the object to the Content Manager Database
Delete	Deletes the object from the Content Manager Database.

Interactive Methods

Most methods allow the programmer to automatically perform some sort of data transformation in Content Manager based on values provided in code. The values may be determined at design-time, or they may be derived from the user at run-time through the custom application's user interface. However, sometimes the integrated application requires that the user interacts directly with one or more Content Manager objects, and therefore requires an Content Manager dialog to be displayed. The Content Manager object model exposes various methods that allow the programmer to invoke standard Content Manager dialogs to be displayed to the user.

These interactive methods must only be called by code running on a client workstation, as most will invoke a modal dialog, which must be explicitly cleared by the user before program execution can continue.

The interactive methods of an object are always identified by the suffix "UI" (for User Interface) and will always take a ParentHWND parameter, which is a handle to the window object that will be the parent of the dialog. (Windows requires this to know how the dialog should behave when the user switches between running applications). In the Visual Basic development environment, the global property hWnd will always contain a handle to the current active window, and can therefore be used as the argument for the ParentHWND parameter. If the parent window handle is not able to be determined, you can pass a "0" instead, in which case Content Manager will place the dialog in front of the current foreground window of the current application. It is also possible to force a null parent for the dialog by passing a handle of "–1". This will force the dialog to be a top level desktop window.

Using the Content Manager Object Model

Database Object

The Database object is the top-level object in the Content Manager object model hierarchy. It is generally the first object to be created when using the Content Manager COM SDK.

Because most objects in Content Manager can only exist in the context of a Database, the Database object is used for accessing and creating all other persistent business objects in the Content Manager COM SDK. These objects are dependent upon the Database object and cannot be created independently.

> **NOTE:** There are certain helper objects in the object model that do not need a Database, such as InputDocument, ExtractDocument, SignatureTool and EnumHelper.

Working with Base Objects

Accessing Persistent Objects

There are generally two ways to access existing persistent objects in the Content Manager Database. The most reliable way is to use the object's URI, as this is guaranteed to uniquely identify the object. The alternative is to use the object's Name – in most cases this is also unique, but the name of an object can change after it is created, whereas the URI cannot.

The Database object has a number of methods for accessing different objects by their URI or Name, all taking the form:

```
Get<object> (LookForValue as Variant) As <object>
```

To instantiate an existing object, you must follow these steps:

1. Declare an object variable of the appropriate object type.
2. Determine the Name or URI of the object to be instantiated.
3. From a Database object, call the appropriate Get<object> method, passing the URI or Name as an argument to the method.
4. If the identifier is valid, the instantiated object will be returned by the method and assigned to the object variable.

The following sample code instantiates an existing record object with a Record Id of "RP95/1".

In Visual Basic

```
In Visual Basic' Declare the object variable
Dim objRecord As TRIMSDK.Record
Dim vntRecId As Variant
' Determine the identifier
vntRecId = "RP95/1"
' Call Get… to instantiate the object
Set objRecord = objTRIM.GetRecord (vntRecId)
' Check that a record with this record number was found
If objRecord Is Nothing Then
        Msgbox "Record ID not found or not accessible due to security."
End If
```

In C#

```
// Determine the identifier
string vntRecId = "RP95/1";
// Call Get… to instantiate the object
TRIMSDK.Record objRecord = db.GetRecord(vntRecId);
// Check that a record with this record number was found
if (objRecord == null)
{
MessageBox.Show("Record ID not found or not accessible due to security.");
}
```

Creating a New Object

All primary persistent objects can be created from the SDK via methods on the Database object. The format of these methods is "New<Objectname>".

A New<object> method returns a new instance of the specified object type. This object contains only default information relating to the object type to begin with, and its properties must be set by code (or by interaction with the user). Calling the "Save" method on the object commits the data to the Database.

The process for creating new objects is therefore as follows:

1. Define an object variable of the type <Object>.
2. On a Database object, call one of the NewObject methods (The return value is the new object).
3. Set the properties of the <Object> variable, or call methods on it to set its data.
4. Call the Save method on the <Object> variable.

The following sample code create a new Keyword (Thesaurus Term) object.

In Visual Basic

```
' Declare the object variable
Dim objKeyword As TRIMSDK.Keyword
' Call New… to instantiate the object
Set objKeyword = objTRIM.NewKeyword
' Set properties
objKeyword.Name = "Example"
objKeyword.TopTerm = True
' Save to the Database
objKeyword.Save
```

In C#

```
// Declare the object variable and call New… to instantiate the object
TRIMSDK.Keyword objKeyword = db.NewKeyword();
// Set properties
objKeyword.Name = "Example";
objKeyword.TopTerm = true;
// Save to the Database
objKeyword.Save();
```

Working With Collections of Base Objects

Collections (of Base Objects) are used to manage related groups of objects of the same type. Collections have several standard methods for iterating through the individual objects, and most have additional methods for selecting objects to be included in the collection based on specific criteria.

When a collection is created it is always empty. You must call methods on the collection to select object items to be included in the collection, based on criteria such as names or URIs. When the collection contains object items, you can call methods that act upon the collection as a whole, such as displaying the collection to the user, making a reference to the collection or printing the items in a Report.

The process for creating and working with collections is as follows:

1. Define an object variable of the type <Objects>.
2. On a Database object, call one of the "Make<Objects>" methods.
3. Add items to the collection using a "Select…" method, or allow the user to search for items using the RefineUI method (if implemented on this collection type).
4. Call methods to manipulate the collection as a group (see table below of common collection methods), if required.
5. To access individual objects in the collection, call ChooseOneUI (user selection), Next (sequential access), or Item (indexed access). Each of these will return an instantiated object of the collection's type.

The following sample code allows the user to choose a single Record Type from all the Record Types in the Database.

In Visual Basic

```
Dim colRecTypes As RecordTypes

Dim objRecType As RecordType

Set colRecTypes = objTRIM.MakeRecordTypes

Call colRecTypes.SelectAll

Set objRecType = colRecTypes.ChooseOneUI(hWnd)
```

In C#

```
TRIMSDK.RecordTypes colRecTypes = db.MakeRecordTypes();

colRecTypes.SelectAll();

int hWnd = Handle.ToInt32();

TRIMSDK.RecordType objRecType = colRecTypes.ChooseOneUI(hWnd);
```

Method	Description
SelectAll	Fill the collection with all the objects of its type from the Database.
SelectByPrefix	Add items to the collection by name prefix.
SelectByUris	Add specific items to the collection. Takes an array of object URIs.
other "Select…" methods	Add items by other criteria. Different collection types will implement different selectors.
RefineUI	Allow the user to select items using a search dialog. Note: Not all collections provide this method.

Working With Child Objects

Child objects are dependents of base objects, and represent either sub-items of the base object (for example, Addresses of a Location) or relationships with other base objects (for example, Keywords attached to a Record).

The only base objects in the Content Manager Object Model that have Child objects are Records, Locations and LookupSets (see the Object Model diagram). The names of Record child objects are prefixed with "Rec", the names of Location child objects are prefixed with "Loc", and the name of the LookupSet child objects are prefixed with "Cds".

The generic process for working with Child objects is slightly different to that for Base objects. Child objects always belong to a collection of "children" that is only able to be instantiated from the parent object. New child objects can be created by calling the New method on the collection, and they can be deleted by calling the Delete method on the child object itself. Any changes to child objects (including additions and removals) are committed to the Database when the parent object is saved.

The process for instantiating a collection of child objects is as follows:

1. Define a collection object variable of the type <ChildObjects>.
2. Set the collection object variable to receive the value of the <ChildObjects> read-only property on an instantiated parent (Record, LookupSet or Location) object.

The following sample code instantiates the collection of Attached Keywords for the Record "RP95/1", and displays them to the user.

In Visual Basic

```
Dim objRecord As Record

Dim colKeywords As RecKeywords


Set objRecord = objTRIM.GetRecord("RP95/1")

Set colKeywords = objRecord.RecKeywords

Call colKeywords.DisplayUI(hWnd)
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord("RP95/1");

TRIMSDK.RecKeywords colKeywords = objRecord.RecKeywords;

int hWnd = Handle.ToInt32();

colKeywords.DisplayUI(hWnd);
```

Editing Child Objects

The process for editing an existing child object is as follows:

1. Define an object variable of the type <ChildObject>
2. From an instantiated child collection, set the child object variable to receive the return value of the GetByUri method, the Item(n) read-only property or the ChooseOne method.
3. Edit the properties of (and/or call methods on) the child object variable.
4. Save the parent object.

The following sample code modifies the contacts for the Record "G96/201", changing contacts of type 'Other' into type 'Addressee'.

In Visual Basic

```
Dim colContacts As RecLocations

Dim objContact As RecLocation

Set objRecord = objTRIM.GetRecord("G96/201")

Set colContacts = objRecord.RecLocations

For i = 0 To colContacts.Count - 1 ' NB collections are zero-based

       Set objContact = colContacts.Item(i)

              If objContact.RecLocType = rlContact _

              and objContact.Subtype = ctOther Then

                     objContact.Subtype = ctAddressee

              End If

Next

Call objRecord.Save
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord("G96/201");

TRIMSDK.RecLocations colContacts = objRecord.RecLocations;

TRIMSDK.RecLocation objContact;

for (int i = 0; i < colContacts.Count; i++)

// NB collections are zero-based

{

      objContact = colContacts.Item(i);

      if (   objContact.RecLocType == rlRecordLocationType.rlContact

             && objContact.Subtype == ctContactType.ctOther)

      {

             objContact.Subtype = ctContactType.ctAddressee;

       }

}

objRecord.Save();
```

Creating New Child Objects

Not all Child collections support creation of new objects. The RecRevisions collection, for example, cannot be explicitly added to because its members are only created through the process of checking in a document as a new revision.

The process for creating a new child object is as follows:

1. Define an object variable of the type <ChildObject>
2. From an instantiated child collection, set the child object variable to receive the return value of the New method.
3. Edit the properties of (and/or call methods on) the child object variable.
4. Save the parent Record or Location.

> **NOTE:** However, that in most cases the Record object also provides 'shortcut' methods as an alternative means of creating new child objects, where the properties of the child object are set through parameters on the method (for example, the AttachRelationship method creates a new RecRelationship child object).
>
> (See the table below for the shortcut methods exposed by the Record interface).

Record Method	Child Object Created
AttachContact	RecLocation

Record Method	Child Object Created
AttachKeyword	RecKeyword
AttachRelationship	RecRelationship
MakeRequest	RecRequest

Deleting Child Objects

Some child objects represent a sub-item of an object (such as a Location Address) that cannot exist independently of the parent, and deleting the child object therefore permanently deletes the sub-item from the Database. However, when you delete a child object that represents a relationship between base items, you are only deleting the relationship, not the item (for example, deleting the RecKeyword "Marine Animals" from the RecKeywords collection belonging to Record "RP95/2" simply detaches the keyword from the record).

The process for deleting a child object is as follows:

1. Define an object variable of the type <ChildObject>
2. From an instantiated child collection, set the child object variable to receive the return value of the GetByUri method, the Item(n) read-only property or the ChooseOne method.
3. Call the Delete method on the child object variable.
4. Save the parent object.

Database Independent Objects

There are a number of objects in the object model that are not dependent on a Database object to be used in the SDK. These objects are:

- InputDocument
- ExtractDocument
- SignatureTool
- EnumHelper

The above of objects are not instantiated through the database object, with a ‘Database.New<object>’ statement. They are simply instantiated on their own, as is any typical object in your IDE.

Object Properties

Overview

The standard data properties of an object are explicitly exposed – that is, there is a named property representing each predefined value of an object. These properties are strongly typed – meaning that each has a specific data type (for example, String, Boolean, Long Integer, Date) and the data variable used to set or get the property value must match that type. An object's properties can be accessed according to the conventions of the automation language you are using.

In Visual Basic (and most other automation languages), an object property is accessed in the following way:

```
strMyValue = objRecord.Title            ' read the Title property

objRecord.Title = strMyValue            ' update the Title property
```

(Some automation languages require property values to be accessed through special methods, in which the property name is prefixed with values such as 'get_' to read and 'put_' to update).

It is also possible to manipulate object properties through generic methods, by using Content Manager's internal property identifiers to specify a property (see the Property Ids section), and a variant data type to hold the property's data value. This technique is discussed in the section on the PropertyDef object.

Reading Properties

You can read the value of an individual Content Manager object property simply by accessing the property by name. Properties always have a specific data type, and therefore your usage of the property should be consistent with the property type. A Record object's Title property, for example, is a String type, whereas the HomeLoc property returns a Location object.

All Base and Child objects also expose a GetProperty method, and you can call this method to retrieve a property value as a Variant type. The method requires that you pass as a parameter the unique property identifier for the property you require.

Updating Properties

You can set or update the value of an object property by assigning a value of the correct data type directly to the property by name.

> **NOTE:** That some named object properties are read-only, and therefore cannot be updated. The reference documentation and the object viewer indicate which properties are read-only.

Most objects also expose a SetProperty method, and you can call this method to update the property value by passing a new value argument as a Variant and the unique property identifier as an integer value.

User-Defined Properties

In Content Manager, the system administrator can define any number of additional User Defined Fields to be associated with Content Manager records. The values of these User Defined Fields can be accessed and updated through the SDK by using the Record object's GetUserField and SetUserField methods.

These methods pass field values as variant data types, and require a FieldDefinition argument to specify the desired field. A FieldDefinition object can be instantiated (by name or Uri) using the GetFieldDefinition method on the Database object.

(See *The FieldDefinition Object* section).

The PropertyDef Object

Generic handling of properties of all Content Manager objects is managed through the PropertyDef object and its associated PropertyDefs collection.

A PropertyDef object manages the unique identifier for an object's property, and provides additional information about the format and structure of the property. It does not, however, contain the data value of the property.

Individual PropertyDef objects are instantiated by specifying a unique internal property Id (see the section on Property Ids), or alternatively (and more easily) by iterating through an instantiated PropertyDefs collection. The collection is instantiated by calling one of the 'Select…' methods exposed by the collection, each of which takes an argument identifying a base object type (a member of the btyBaseObjectTypes enumeration). These methods add property definitions to the collection for the specified object type, and provide options to select all properties for the object type, all properties available to the View Pane (or those included on the view pane by default), all modifiable properties or all properties for a given subgroup.

The following sample code demonstrates the use of a PropertyDefs collection, which is in this case instantiated using the method SelectViewPaneItems, passing the Record object type identifier. In a loop, a PropertyDef object is used to iterate the collection, and the program outputs each property's Caption and the string representation of its data value.

In Visual Basic

```
Private Sub PrintProperties(objRecord As Record)

        Dim objProp As New PropertyDef

        Dim colProps As New PropertyDefs


        Call colProps.SelectViewPaneItems(btyRecord)

        For i = 0 To colProps.Count - 1

                Set objProp = colProps(i)

                Debug.Print objProp.GetCaption(objRecord.Database) &  _

                " : " & objRecord.GetPropertyAsString(objProp)

        Next

End Sub
```

In C#

```
private void PrintProperties(Record objRecord)
{
        TRIMSDK.PropertyDef objProp = new TRIMSDK.PropertyDef();
        TRIMSDK.PropertyDefs colProps = new TRIMSDK.PropertyDefs();
       // C# requires specification of default parameters
        colProps.SelectViewPaneItems(btyBaseObjectTypes.btyRecord,false);
        for (int i = 0; i < colProps.Count; i++)
        {
               objProp = colProps[i];
               Console.WriteLine("{0}:{1}", objProp.GetCaption(objRecord.Database,
false),
            objRecord.GetPropertyAsString(objProp, sdStringDisplayType.sdDefault,
false));;
         }
}
```

The FieldDefinition Object

Content Manager allows a large number of User Defined Fields to be assigned to records. Therefore User Defined Fields cannot be interrogated using normal named properties of the record object. Instead, accessing User Defined Fields is carried out using a dedicated object for managing these fields, the FieldDefinition object, and its associated FieldDefinitions collection.

The Record object has a pair of methods for manipulating User Defined Fields, GetUserField and SetUserField. Each method takes a populated FieldDefinition object as a parameter.

Acting on Content Manager Events

There are a number of interfaces allowing the programmer to run custom code in response to events that occur in Content Manager. They are:

- RecordAddIn
- FieldAddIn
- BaseObjectAddIn
- EventProcessor

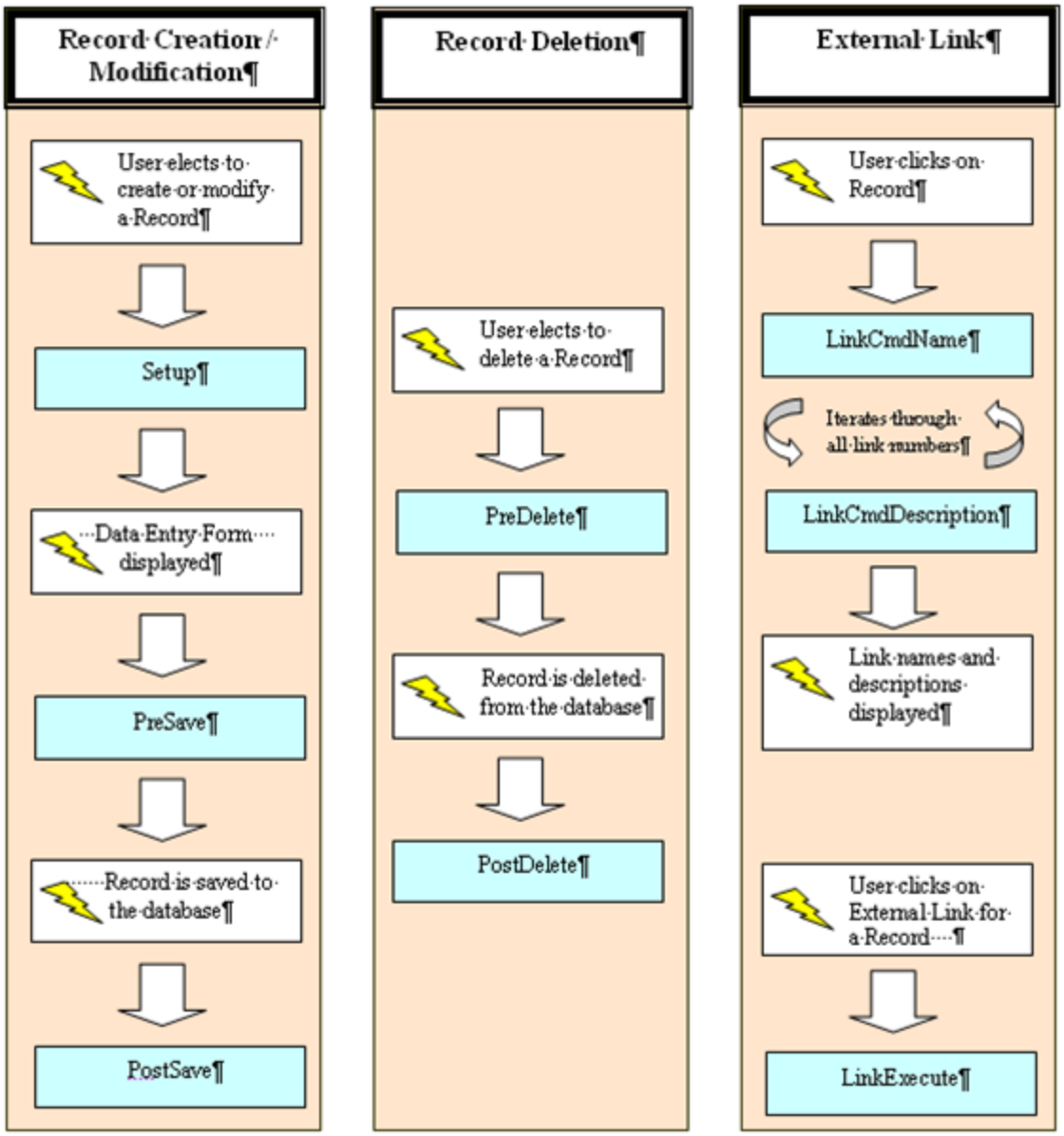
RecordAddIn

The RecordAddIn gives access to generic entry points in various processes for the Record. Methods, within which custom code may be placed, are shown below in blue.

Although the RecordAddIn class is defined in the TRIMSDK Type Library (along with all the other Content Manager COM SDK object classes), it is a special class, in that it is only an interface definition, and the programmer must implement the interface to act upon method calls made by Content Manager. (All other classes in the Type Library represent interfaces implemented by Content Manager, and the programmer can call the methods on these interfaces).

The ErrorMessage property of the FieldAddIn (not shown in the diagram below) provides some functionality for catching and displaying error messages to the user. If the PreSave or PreDelete return false, the ErrorMessage property will be called and populated with the corresponding error message. The SDK programmer may also call the ErrorMessage property explicitly in code and populate it with a customized error message.

It should be noted that none of the methods of the RecordAddIn provide a ParentHWnd parameter (a handle to the user's current window). is not advisable to use dialogs in situations where no ParentHWnd parameter is provided, as without a handle to the user's window, the dialogs may be popping up on the server machine with nobody to answer them.



FieldAddIn

The FieldAddIn provides access to generic entry points in processes involving User Field modification. Methods, within which custom code may be placed, are shown below in blue.

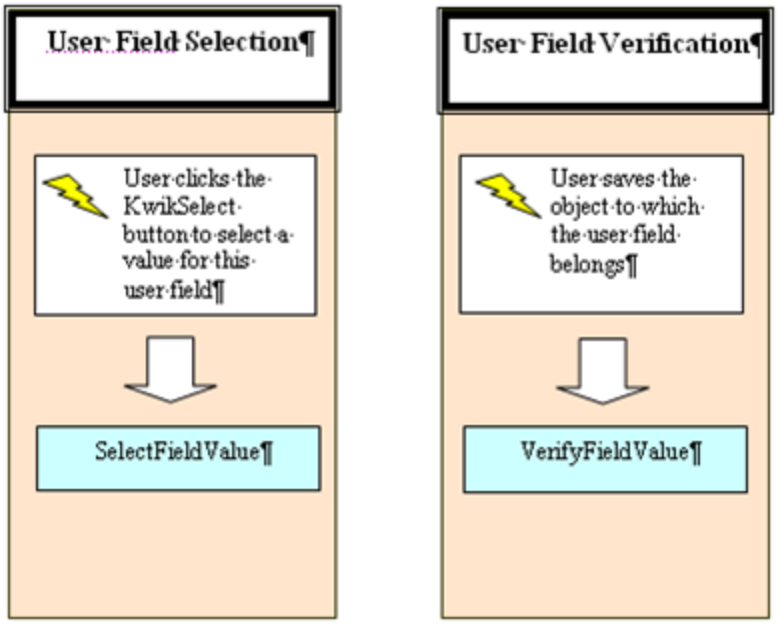
Although the FieldAddIn class is defined in the TRIMSDK Type Library (along with all the other Content Manager COM SDK object classes), it is a special class, in that it is only an interface definition, and the programmer must implement the interface to act upon method calls made by Content Manager. (All other classes in the Type Library represent interfaces implemented by Content Manager, and the programmer can call the methods on these interfaces).

The ErrorMessage property of the FieldAddIn (not shown in the diagram below) provides some functionality for catching and displaying error messages to the user. If the VerifyFieldValue method returns ‘false’, the ErrorMessage property will be called and populated with the corresponding error

message. The SDK programmer may also call the ErrorMessage property explicitly in code and populate it with a customized error message.

Another feature of the FieldAddIn is the ParentHWnd parameter provided to the SelectFieldValue method. This provides a handle to the user's current window and enables custom dialogs to be displayed to the user (even when the FieldAddIn is running on the server). It is not advisable to use dialogs in situations where no ParentHWnd parameter is provided, as without a handle to the user's window, the dialogs may be popping up on the server machine with nobody to answer them.

It is often desirable to access the object to which the user field undergoing the selection or verification belongs. This may be done using the FieldDefinition method 'GetCurrentObject' for the user field passed to the FieldAddIn methods. In contrast, in the BaseObjectAddIn, the parent object to the user field is passed as a parameter to the SelectFieldValue and VerifyFieldValue methods, so when using the BaseObjectAddIn using the 'GetCurrentObject' method is not necessary.



BaseObjectAddIn

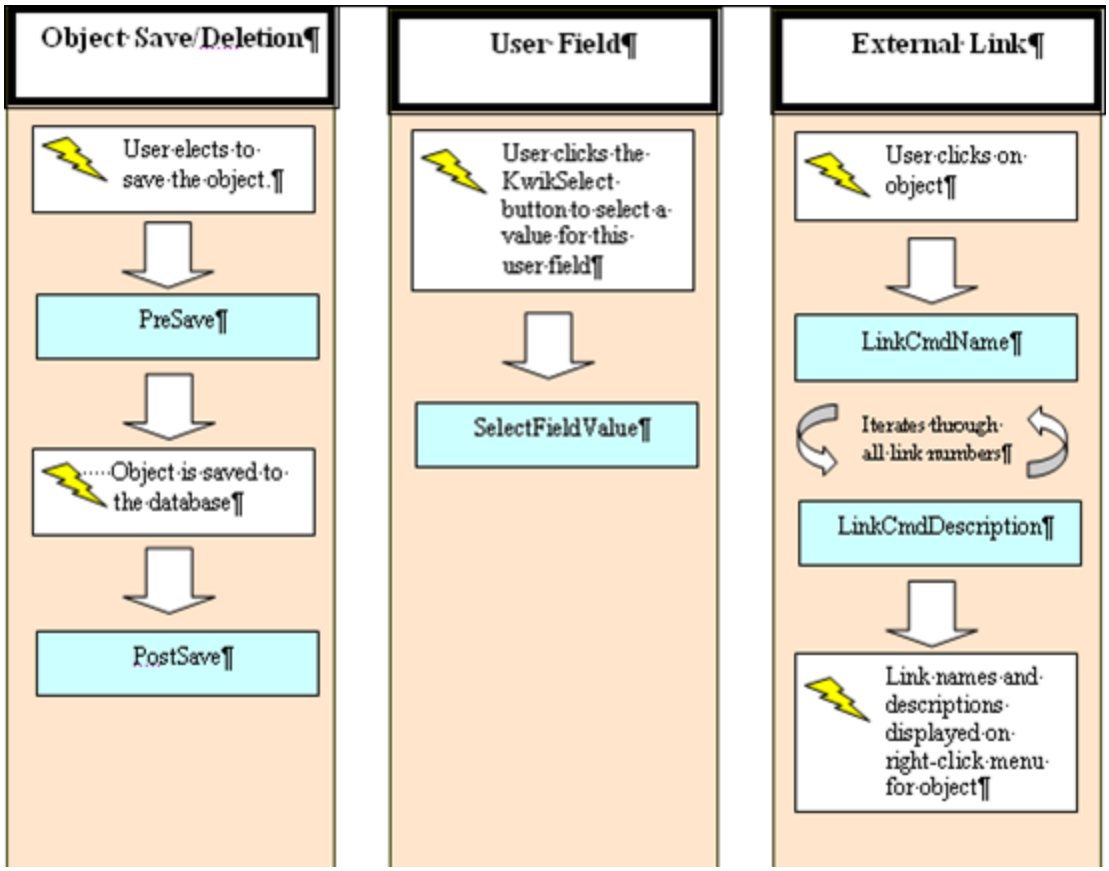
The BaseObjectAddIn is a generic AddIn that can respond to entry points within the Save or Delete process for any Content Manager Object. Methods, within which custom code may be placed, are shown below in blue.

Although the BaseObjectAddIn class is defined in the TRIMSDK Type Library (along with all the other Content Manager COM SDK object classes), it is a special class, in that it is only an interface definition, and the programmer must implement the interface to act upon method calls made by Content Manager. (All other classes in the Type Library represent interfaces implemented by Content Manager, and the programmer can call the methods on these interfaces).

The ErrorMessage property of the BaseObjectAddIn (not shown in the diagram below) provides some functionality for catching and displaying error messages to the user. If any of the PreSave, PreDelete, or VerifyFieldValue methods return 'false', the ErrorMessage property will be called and populated with the corresponding error message. The SDK programmer may also call the ErrorMessage property explicitly in code and populate it with a customized error message.

Another feature of the BaseObjectAddIn is the ParentHWnd parameter provided to the SelectFieldValue method. This provides a handle to the user's current window and enables custom dialogs to be displayed to the user (even when the FieldAddIn is running on the server). It is not advisable to use dialogs in situations where no ParentHWnd parameter is provided, as without a handle to the user's window, the dialogs may be popping up on the server machine with nobody to answer them.

The SelectFieldValue and VerifyFieldValue methods of the BaseObjectAddIn differ from the TRIMFieldAddIn in that a the object to which the user field belongs is passed to the method. This enables the programmer to access other properties and fields of the parent object, which may be useful in selecting/verifying the value of the user field. It may also be used to set other properties and fields of the parent object at the time a value for the user field is selected or verified.



EventProcessor

The EventProcessor interface is an AddIn that responds to the processing of an Content Manager Event by the Content Manager Event Processor. Methods, within which custom code may be placed, are shown below in blue.

Although the TRIMEventProcessor class is defined in the TRIMSDK Type Library (along with all the other Content Manager COM SDK object classes), it is a special class, in that it is only an interface definition, and the programmer must implement the interface to act upon method calls made by Content Manager. (All other classes in the Type Library represent interfaces implemented by Content Manager, and the programmer can call the methods on these interfaces).

The range of events processed by the Event Processor is wide, but includes actions such as users logging on and off, records being created or modified, documents being accessed, and various security-related events.

It is important to note that events are processed by the Event Processor (and the ProcessEvent method called) at a later time to that at which the event occurred. As a result the SDK programmer cannot rely on the object to which the event occurred still being in the state indicated by the event, since the ProcessEvent method is not called in direct response to the event occurring. This must be taken into account in the design of a program implementing the EventProcessor interface. For example, if a hold is added to a record, the event ‘evHoldAdded’ is fired. However, by the time the EventProcessor comes to process this ‘evHoldAdded’ event, the Hold may have already been removed from the record, so you cannot rely on the record being under a Litigation Hold at the time the ‘evHoldAdded’ event is processed.

Common Scenarios – Code Samples

In this section, various common programming scenarios are discussed, with sample code to illustrate how different tasks can be achieved.

Connecting to a Database

The Database object manages each client's connection to a Database, and in doing so it authenticates the current user (from their network login) and applies their Content Manager security profile when accessing any Content Manager data. The Database object's Connect method will attempt to connect the user to their default Database. It does not require any parameters.

In Visual Basic

```
Dim objTRIM as TRIMSDK.Database

Dim colDBs as New TRIMSDK.Databases

If colDBs.Count > 1 Then

       ' Let user select a Database in a dialog

       Set objTRIM = colDBs.ChooseOneUI(hWnd)

End If

objTRIM.Connect
```

In C#

```
TRIMSDK.Database db = null;

TRIMSDK.Databases colDBs = new TRIMSDK.Databases();

if (colDBs.Count > 1)

{

int hWnd = Handle.ToInt32();

        // Let user select a Database in a dialog

        db = colDBs.ChooseOneUI(hWnd);

}

if ( db != null )

{

       db.Connect();

}
```

Similarly, if a particular named Database is required, this could also be selected programmatically from the Databases collection.

In Visual Basic

```
For i = 0 to colDBs.Count -1
	Set objTRIM = colDBs.Item(i)
	If objTRIM.Name = "MyTRIM" Then
		objTRIM.Connect
		Exit For
	End If
Next
```

In C#

```
for ( int i=0;  i < colDBs.Count; i++)
{
	db = colDBs.Item(i);
	if (db.Name == "MyTRIM")
	{
		db.Connect();
		break;
	}
}
```

Connecting to a default database

The following sample code demonstrates connecting to a Content Manager Database, when the Content Manager Installation is a default database, or there is only one database.

In Visual Basic

```
Dim objTRIM As TRIMSDK.Database
Set objTRIM = New TRIMSDK.Database
objTRIM.Connect
```

In C#

```
TRIMSDK.Database db = new TRIMSDK.Database();

db.Connect();
```

> **NOTE:** Calling the Connect method in this way is optional. If the Database object is not connected when a method requiring a Database service is called, Content Manager will automatically attempt a connection.

Connecting to a specific database

The following sample code demonstrates connecting to a specific Content Manager Database by setting the database id.

In Visual Basic

```
Dim objTRIM As TRIMSDK.Database

Set objTRIM = New TRIMSDK.Database

objTRIM.Id = “45”  ‘45 is the id of the  Content Manager Demonstration Database

objTRIM.Connect
```

In C#

```
TRIMSDK.Database db = new TRIMSDK.Database();

db.Id = “45”;  //45 is the id of the  Content Manager Demonstration Database

db.Connect();
```

> **NOTE:** Calling the Connect method in this way is optional. If the Database object is not connected when a method requiring a Database service is called, Content Manager will automatically attempt a connection.

Allowing the user to choose a Database

Instantiates a Database selected by the user from a list of Databases.

In Visual Basic

```
Dim p_TRIMDatabaseCollection As TRIMSDK.Databases

Dim p_TRIMDatabase As TRIMSDK.Database

' Instantiate collection of valid  Content Manager Databases

Set p_TRIMDatabaseCollection = New TRIMSDK.Databases

' Display the list of Databases.

' Assign the selection to the modular level  Content Manager Database variable

Set p_TRIMDatabase = p_TRIMDatabaseCollection.ChooseOneUI(hWnd)

If p_TRIMDatabase Is Nothing Then

        Debug.Print "User Cancelled!"

Else

        Debug.Print " Content Manager Database " & p_TRIMDatabase.Name & " is
Connected(T/F) - " & p_TRIMDatabase.IsConnected

End If

' Release object

Set p_TRIMDatabaseCollection = Nothing
```

In C#

```
// Instantiate collection of valid  Content Manager Databases
TRIMSDK.Databases dbCol = new TRIMSDK.Databases();
// Display the list of Databases.
// Assign the selection to the modular level  Content Manager Database variable
int hWnd = Handle.ToInt32();
TRIMSDK.Database db = dbCol.ChooseOneUI(hWnd);
if (db == null)
{
        Console.WriteLine("User Cancelled!");
}
else
{
        Console.WriteLine( "TRIM Database " + db.Name + " is Connected(T/F) - " +
db.IsConnected);
}
// Release object
dbCol = null;
```

Showing and editing the properties of the Content Manager Database object

The following sample code demonstrates one method of customising the database properties in the SDK.

In Visual Basic

```
Private m_TRIMDatabase As TRIMSDK.Database

If m_TRIMDatabase.PropertiesUI(hWnd) Then

        If m_TRIMDatabase.Verify Then

                m_TRIMDatabase.Save

        Else

                MsgBox "Error Saving Database properties " & m_
TRIMDatabase.ErrorMessage, vbExclamation

        End If

Else

        Debug.Print "User Cancelled"

End If
```

In C#

```
private TRIMSDK.Database db = new TRIMSDK.Database();

int hWnd = Handle.ToInt32();

if (db.PropertiesUI(hWnd))

{

       if (db.Verify(false))

       {

             db.Save();

       }

       else

       {

            MessageBox.Show("Error Saving Database properties " + db.ErrorMessage,"
",System.Windows.Forms.MessageBoxButtons.OK,
System.Windows.Forms.MessageBoxIcon.Exclamation);

       }

}

else

{

       Console.WriteLine( "User Cancelled");

}
```

Accessing a Record

To read information stored on records in a Content Manager Database, the API programmer must first determine how to access the records required. If a particular record's internal or external unique identifier is known, the associated record can be accessed directly and efficiently using the GetRecord method. (If neither of these unique identifiers are known, it will be necessary to construct a search. This is covered in the section Searching for Records).

Getting a Record by Record Number

Every record in Content Manager has a unique Record Number. This follows a pattern defined by the Record Type and can be manually entered by the user or set to be automatically generated by Content Manager. Although the commonly used term is 'number', it is more correctly an identifier, as it is a string that may contain alphanumeric characters. This string is accessible through the Record object's Number property.

The Record Number can be used as the argument to be passed to the Database object's GetRecord method, which takes a variant for the unique identifier and returns a pointer to the instantiated Record.

The following sample code instantiates the record 2002/0059 by record number:

In Visual Basic

```
Set objRecord = objTRIM.GetRecord ("2002/0059") ' instantiate by number
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord ("02/59"); //instantiate by number
```

> **NOTE:** Content Manager stores the Record Number in two formats. The expanded format (for example, "2002/0059") is held in the LongNumber property, and the compressed format (for example, "02/59") is held in the Number property. Both can be passed to the GetRecord method.

Getting a Record by URI

The Unique Row Identifier or URI of a record is an internal unique number that is transparent to the everyday user of Content Manager. It is the primary key on the TSRECORD table in the Database and provides an internal unique identifier for every record.

To instantiate a record by its URI, you can pass the numeric URI as the argument to the Database object's GetRecord method.

The following sample code instantiates the record by URI:

In Visual Basic

```
Set objRecord = objTRIM.GetRecord (130) ' instantiate by URI
```

In C#

```
objRecord = db.GetRecord (130); // instantiate by URI
```

Once an instantiated record object has been returned by the GetRecord method, the programmer can access properties and call methods on the object. These are discussed in the following subsections.

Reading Basic Properties

Most of the metadata directly associated with a record is exposed through properties on the Record interface. Most properties return primitive data types (strings, numbers or dates) and can be interrogated directly. The meanings of these properties are generally self-evident from their names, but are also given in the object browser and in the reference section.

Examples of basic readable properties of a record are:

Property	Example value
Number	"G1997/0770"
Title	"Greenhouse Journal of Global Warming - Dugong Habitats"
DateCreated	#8/20/1997#
ExternalId	"GJGW 97PB"
AccessionNbr	5617

In Visual Basic

```
Dim objRec As Record

Set objRec = objTRIM.GetRecord ("G97/770")

If objRec.AccessionNbr > 5000 and objRec.DateCreated < #01/01/2000# Then

       Msgbox objRec.Title, , "Record " & objRec.Number

End If
```

In C#

```
TRIMSDK.Database db = new TRIMSDK.Database();

TRIMSDK.Record objRecord = db.GetRecord ("G97/770");

DateTime date = new DateTime(2000,01,01);

if (objRecord.AccessionNbr > 5000

       && objRecord.DateCreated < date)

{

       MessageBox.Show (objRecord.Title, "Record " + objRecord.Number);

}
```

Accessing Related Objects

Many attributes of a Content Manager record represent other objects, such as the RecordType, Classification and Container attributes. These are properties where the data type of the property is an object interface reference.

The following sample code instantiates a record object (in variable objRecord) and then assigns its Container to another variable (objContainer).

In Visual Basic

```
Dim objRecord As Record

Dim objContainer As Record

Set objRecord = objTRIM.GetRecord ("G99/15")

Set objContainer = objRecord.Container ' objContainer is now 97/1004
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord ("G99/15");

TRIMSDK.Record objContainer = objRecord.Container; ' objContainer is now 97/1004
```

Accessing Record Location Information

A Content Manager record has various properties concerning related location information. These properties of a Record all return an instantiated Location object:

- CurrentLoc – Current (Assignee) location of the record
- HomeLoc – Normal location of the record
- OwnerLoc – Location of the Owner or responsible Organization for the record
- AuthorLoc – Person who authored the electronic document
- CreatorLoc – Person who registered the record in Content Manager

- AddresseeLoc – Person to whom the record is addressed
- PrimaryContactLoc – The main contact person (or organization) for the record.

The following sample code shows how to access the properties and methods of these location objects by creating and instantiating them:

In Visual Basic

```
Dim objRec as Record

Dim objLoc as Location

Set objRec = objTRIM.GetRecord ("2002/0059") ' instantiate the record

Set objLoc = objRec.AuthorLoc ' get the author location object

Msgbox "Author's name is: " & objLoc.FormattedName
```

In C#

```
// instantiate the record

TRIMSDK.Record objRecord = db.GetRecord ("G99/15");

// get the author location object

TRIMSDK.Location objLoc = objRecord.AuthorLoc;

MessageBox.Show ("Author's name is:" + objLoc.FormattedName);
```

Updating Records

So far we have only considered the methods for reading information from records in Content Manager. The COM SDK also allows you to update Content Manager records, either by updating the values of properties on a given record object, or by calling methods on the record.

Updating properties is the simplest way to modify the metadata of a record. You simply assign a new value of the correct data type to the named property of the object. Field-level verification is carried out, and an error will be raised if the property update is invalid (see also Verifying and Error Trapping section). For more complicated types of update to a record, you must generally call methods that instruct Content Manager to modify the record, based on arguments passed.

Modifying Properties

The simplest way to update data in an Content Manager record is to modify the named properties on the Record object. This can only be done on properties that are not marked as read-only. This includes most of the Date properties, certain Location properties (AuthorLoc, AddresseLoc and OtherLoc) and miscellaneous properties such as External Id, Priority, Accession Number and Foreign Barcode.

In Visual Basic

```
Set objRecord = objTRIM.GetRecord(30)

objRecord.Title = "New title for this record"

objRecord.DateDue = Date + 10 ' Due in ten days

objRecord.DatePublished = #20/05/2002#

Set objRecord.AuthorLoc = objTRIM.CurrentUser
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord(30);

objRecord.Title = "New Title for this record";

objRecord.DateDue = DateTime.Today.AddDays(10);

DateTime datePub = new DateTime(2002,05,20);

objRecord.DatePublished = datePub;

objRecord.AuthorLoc = db.CurrentUser;
```

Calling Update Methods

To update other data on a record where read-write properties are not available, you must call a method instead. Update methods generally begin with the prefix 'Set…' and they include a parameter for the new data value you wish to apply.

In Visual Basic

```
Call objRecord.SetCurrentLocation(objMyUnitLoc);
```

In C#

```
TRIMSDK.Location objMyUnitLoc = db.CurrentUser;

objRecord.SetCurrentLocation(objMyUnitLoc,DateTime.Today);
```

In many cases other parameters can be specified that control the behavior of the update:

In Visual Basic

```
' Set Current location to me, effective from yesterday

Call objRecord.SetCurrentLocation(objTRIM.CurrentUser, Date - 1)
```

In C#

```
// Set Current location to me, effective from yesterday

DateTime yesterday = DateTime.Today.AddDays(-1);

objRecord.SetCurrentLocation(db.CurrentUser, yesterday);
```

Updating Properties Using SetProperty

To update a record's properties where the internal identifier of the property is known (see Property Ids), you can use the SetProperty method. This requires passing the property identifier and a variant containing the data value.

In Visual Basic

```
' Set the title (property id=3)

Call objRecord.SetProperty(3, "Barrier Reef manatee population figures")
```

In C#

```
// Set the title (property id=3)

objRecord.SetProperty(3, "Barrier Reef manatee population figures");
```

Verifying

When a record object is modified via the SDK, there are two levels of verification that must be carried out before the changes can be committed to the Database.

The first is field-level verification, which checks that the change to an individual property is legal. An example would be to check that a Record’s Date Created is not in the future. If a property update cannot be carried out because of field-level verification, the attempt to set the property will cause a run-time error to be raised and the update will not be carried out.

The second level of validation is object-level verification (sometimes called cross-field verification). This checks that the values of all fields on the object are consistent with each other. An example of object-level verification would be that the Date Registered is not earlier than the Date Created. Object-level verification may be performed by the object’s Verify method. It is also carried out automatically whenever the object is saved.

Object-level verification for a single property may be performed by the base object’s VerifyProperty method. This checks that the value of a nominated property is consistent with all other current values for the object. The VerifyProperty method also sn optional the capability to fail if the property is mandatory and has not been set.

The Verify Method

The Record object (and every other base object) has a Verify method. This can be called to perform object-level verification prior to saving the object. The method returns false if there are any errors in the state of the object, and the error description will be stored in the object's ErrorMessage property. If there are no errors, the method returns true and the Verified property (see The Verified Property) is set to true.

The method contains an optional parameter FailOnWarnings which, if set to true, will cause the Verify method to check for warning conditions as well as error conditions, and to fail if a warning is encountered.

In Visual Basic

```
If Not objRecord.Verify(True) Then
	Msgbox objRecord.ErrorMessage,,"Verify Failed"
Else
	objRecord.Save
End If
```

In C#

```
if (! objRecord.Verify(true))
{
	MessageBox.Show(objRecord.ErrorMessage,"Verify Failed");
}
else
{
	objRecord.Save();
}
```

If it is not called explicitly in code, the Verify method will be automatically called before an object is saved and if verification fails it will not be saved. This ensures that data cannot become corrupted and that business rules are observed when using the SDK, just as they are for users of the Content Manager Client interface.

The Verified Property

Base objects also have a Verified Boolean read-only property, which is false whenever the object is instantiated. It is set to true when the Verify method confirms that it is in a legal state to be saved to the Database.

The following sample code demonstrates how the Verified property changes according to the state of the object.

In Visual Basic

```
' To demonstrate the Verified property

Dim objTRIM As TRIMSDK.Database

Set objTRIM = New TRIMSDK.Database

' Instantiate the Record

Dim objRecord As TRIMSDK.Record

Dim msg As String

Dim oldTitle As String

Set objRecord = objTRIM.GetRecord("02/59")

msg = "The Record object has just been instantiated.  Verified property is set to: "
& objRecord.Verified

MsgBox (msg)

' Verify the Record, with default FailOnWarnings = false

objRecord.Verify (False)

msg = "The Record has just been verified.  Verified property is set to: " &
objRecord.Verified

MsgBox (msg)

oldTitle = objRecord.Title

msg = "Would you like to change the title of the Record?"

If MsgBox(msg, vbYesNo, vbQuestion) = VbMsgBoxResult.vbYes Then

      objRecord.Title = "new Title"

      msg = "The title of the Record has just been changed.  The Record has not yet
been checked for internal consistency.  Verified property is set to: " &
objRecord.Verified

      MsgBox (msg)

Else

      msg = "No changes have been made, so the Record is still internally
consistent.  Verified property is set to: " & objRecord.Verified

      MsgBox(msg)

End If

' now save the changes if the object is verified

If Not objRecord.Verified Then

       If objRecord.Verify() Then

              objRecord.Save
```

```
            msg = "The changes made to the Record have been verified, and it has
just been saved (so the changes are now committed to the Database).  The Verified
property is now set to: " & objRecord.Verified

            MsgBox (msg)

      Else

            msg = "Record Verify failed:" & objRecord.ErrorMessage & ".  Because
of this, it has not been saved."

            MsgBox (msg)

      End If

Else

      msg = "Record was verified, so there were no changes to save.")

      MsgBox(msg)

End If

MsgBox ("Reverting back to original title of record...")

objRecord.Title = oldTitle

objRecord.Save
```

In C#

```
// To demonstrate the Verified property

// Instantiate the Record

TRIMSDK.Record objRecord = db.GetRecord("02/59");

MessageBox.Show("The Record object has just been instantiated.  Verified property is
set to: " + objRecord.Verified);

// Verify the Record, with default FailOnWarnings = false

objRecord.Verify(false);

MessageBox.Show("The Record has just been verified.  Verified property is set to: "
+ objRecord.Verified);

string oldTitle = objRecord.Title;

if (MessageBox.Show("Would you like to change the title of the Record?","",
MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)

{

       objRecord.Title = "new Title";

       MessageBox.Show("The title of the Record has just been changed.  The Record
has not been checked for internal consistency.  Verified property is set to: " +
objRecord.Verified);

}

else

{

MessageBox.Show("No changes have been made, so the Record is still internally
consistent.  Verified property is set to: " + objRecord.Verified);

}

// now save the changes if the object is verified

if (! objRecord.Verified)

{

       if (objRecord.Verify(false))

       {

              objRecord.Save();

              MessageBox.Show("The changes made to the Record have been verified,
and it has just been saved (so the changes are now committed to the Database).  The
Verified property is now set to: " + objRecord.Verified);

       }

       else
```

```
	{
		MessageBox.Show("Record Verify failed:" + objRecord.ErrorMessage + ". 
Because of this, it has not been saved.  The Verified property is now set to: " + 
objRecord.Verified);
	}
}
else
{
	MessageBox.Show("Record was verified, so there were no changes to save.  The 
Verified property is now set to: " + objRecord.Verified);
}
MessageBox.Show("Reverting back to original title of record...");
objRecord.Title = oldTitle;
objRecord.Save();
```

The Verified Property

Base objects also have a Verified Boolean read-only property, which is false whenever the object is instantiated. It is set to true when the Verify method confirms that it is in a legal state to be saved to the Database.

The following sample code demonstrates how the Verified property changes according to the state of the object.

In Visual Basic

```
' To demonstrate the Verified property

Dim objTRIM As TRIMSDK.Database

Set objTRIM = New TRIMSDK.Database

' Instantiate the Record

Dim objRecord As TRIMSDK.Record

Dim msg As String

Dim oldTitle As String

Set objRecord = objTRIM.GetRecord("02/59")

msg = "The Record object has just been instantiated.  Verified property is set to: "
& objRecord.Verified

MsgBox (msg)

' Verify the Record, with default FailOnWarnings = false

objRecord.Verify (False)

msg = "The Record has just been verified.  Verified property is set to: " &
objRecord.Verified

MsgBox (msg)

oldTitle = objRecord.Title

msg = "Would you like to change the title of the Record?"

If MsgBox(msg, vbYesNo, vbQuestion) = VbMsgBoxResult.vbYes Then

      objRecord.Title = "new Title"

      msg = "The title of the Record has just been changed.  The Record has not yet
been checked for internal consistency.  Verified property is set to: " &
objRecord.Verified

      MsgBox (msg)

Else

      msg = "No changes have been made, so the Record is still internally
consistent.  Verified property is set to: " & objRecord.Verified

      MsgBox(msg)

End If

' now save the changes if the object is verified

If Not objRecord.Verified Then

       If objRecord.Verify() Then

              objRecord.Save
```

```
            msg = "The changes made to the Record have been verified, and it has
just been saved (so the changes are now committed to the Database).  The Verified
property is now set to: " & objRecord.Verified

            MsgBox (msg)

      Else

            msg = "Record Verify failed:" & objRecord.ErrorMessage & ".  Because
of this, it has not been saved."

            MsgBox (msg)

      End If

Else

      msg = "Record was verified, so there were no changes to save.")

      MsgBox(msg)

End If

MsgBox ("Reverting back to original title of record...")

objRecord.Title = oldTitle

objRecord.Save
```

In C#

```
// To demonstrate the Verified property

// Instantiate the Record

TRIMSDK.Record objRecord = db.GetRecord("02/59");

MessageBox.Show("The Record object has just been instantiated.  Verified property is
set to: " + objRecord.Verified);

// Verify the Record, with default FailOnWarnings = false

objRecord.Verify(false);

MessageBox.Show("The Record has just been verified.  Verified property is set to: "
+ objRecord.Verified);

string oldTitle = objRecord.Title;

if (MessageBox.Show("Would you like to change the title of the Record?","",
MessageBoxButtons.YesNo, MessageBoxIcon.Question) == DialogResult.Yes)

{

       objRecord.Title = "new Title";

       MessageBox.Show("The title of the Record has just been changed.  The Record
has not been checked for internal consistency.  Verified property is set to: " +
objRecord.Verified);

}

else

{

MessageBox.Show("No changes have been made, so the Record is still internally
consistent.  Verified property is set to: " + objRecord.Verified);

}

// now save the changes if the object is verified

if (! objRecord.Verified)

{

       if (objRecord.Verify(false))

       {

              objRecord.Save();

              MessageBox.Show("The changes made to the Record have been verified,
and it has just been saved (so the changes are now committed to the Database).  The
Verified property is now set to: " + objRecord.Verified);

       }

       else
```

```
	{
		MessageBox.Show("Record Verify failed:" + objRecord.ErrorMessage + ". 
Because of this, it has not been saved.  The Verified property is now set to: " + 
objRecord.Verified);
	}
}
else
{
	MessageBox.Show("Record was verified, so there were no changes to save.  The 
Verified property is now set to: " + objRecord.Verified);
}
MessageBox.Show("Reverting back to original title of record...");
objRecord.Title = oldTitle;
objRecord.Save();
```

Trapping Run-Time Errors

It is up to the programmer to determine how they wish to deal with possible errors when updating an object. However, they must be aware that error checking takes place even when directly updating properties, so it will be necessary to provide some error-trapping code to prevent run-time errors being displayed to the user if there is a possibility of errors being raised.

Saving the Record to the Database

All of the update methods and property changes made through the Record interface are only applied to the object in memory. The changes are not committed to the TRIM Database until the object is saved.

Calling the Save method on the record object will commit the changes to the Database, applying all updates since the object was instantiated (or since it was last saved).

> **NOTE:** That if the record has not been verified, Save will automatically call the Verify method and will only commit the changes if the verification succeeds.

In Visual Basic

```
Set objRecord = objTRIM.GetRecord("G97/770")
With objRecord
       .Title = .Title & " plus New Part of Title"
       .DateDue = #1/1/2003#
       Set .AuthorLoc = objTRIM.CurrentUser
       Call .Save  ' commit all these changes to the Database
End With
```

In C#

```
TRIMSDK.Record objRecord = db.GetRecord("G97/770");
objRecord.Title = objRecord.Title + " plus New Part of Title";
DateTime dateDue = new DateTime(2003,1,1);
objRecord.DateDue = dateDue;
objRecord.AuthorLoc = db.CurrentUser;
objRecord.Save();  // commit all these changes to the Database
```

Searching for Records

One of the most powerful features of TRIM is the wide range of search criteria that can be applied to select records from the Database. The SDK has many features available for creating complex and sophisticated searches, yet it can also be used with a minimum of code.

The RecordSearch object enables TRIM records to be retrieved by creating a search expression from a number of search clauses, and has methods to navigate the records that meet the search criteria. The RecordSearch object also allows boolean and, or and not relationships to logically combine search clauses, and setting filters and sort criteria. The object also has file functions for saving searches to or loading from disk.

To set the search criteria for a record search, you can either call search clause methods explicitly, or display the TRIM search dialog to allow the user to specify the search criteria, or a combination of the two.

The process of searching for records via the SDK is as follows:

1. Construct a RecordSearch object
2. Add a search clause
3. Add additional clauses and combine them with logical operators (optional)
4. Apply Record Type filters (optional)
5. Display the criteria to the user (optional)
6. Execute the search query

7. Process the results sequentially, or
8. Copy the results to a record collection.

In Visual Basic

```
Dim objSearch As RecordSearch
Dim colRecords As Records
' Construct a new search object
Set objSearch = objTRIM.NewRecordSearch
' Search for "reef" in record titles
Call objSearch.AddTitlewordClause("reef")
' Hold the results in a collection
Set colRecords = objSearch.GetRecords
```

In C#

```
// Construct a new search object
TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();
// Search for "reef" in record titles
objSearch.AddTitleWordClause("reef");
// Hold the results in a collection
TRIMSDK.Records colRecords = objSearch.GetRecords();
```

Creating a RecordSearch Object

Like any other object, the RecordSearch object must be constructed by the Database object, in this case using the NewRecordSearch method. A RecordSearch object is a temporary object, and therefore does not need to be instantiated from the Database (the exception to this is Saved Searches, which will be covered later).

In Visual Basic

```
Dim objSearch As RecordSearch ' declare the search object
Set objSearch = objTRIM.NewRecordSearch ' make the object
Call objSearch.EditQueryUI(hWnd) ' call methods on the object…
```

In C#

```
// declare & make the search object

TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();

int hWnd = Handle.ToInt32();

objSearch.EditQueryUI(hWnd); // call methods on the object…
```

Adding a Search Clause

Once you have created the search object, you must then add at least one search clause before it can be executed to return results. There are many different search clauses available; the full list can be found in the Reference section.

The following sample code retrieves records that contain the word "reef" within the title, you could add a Title Word clause passing the argument "reef".

In Visual Basic

```
objSearch.AddTitlewordClause("reef") ' search for titles with "reef"
```

In C#

```
objSearch.AddTitleWordClause("reef"); //search for titles with "reef"
```

To retrieve records that were created since January 1, 2001, you would add a Date Created clause passing the arguments "1/1/2001" and the current date, as follows:

In Visual Basic

```
objSearch.AddDateCreatedClause(#01/01/2001#, Date)
```

In C#

```
System.DateTime dateCreated = new DateTime(2001,01,01);

objSearch.AddDateCreatedClause(dateCreated, DateTime.Today);
```

You can build search criteria by calling multiple methods, and applying specific logical relationships, using the Boolean operators, as described below.

Boolean Operators - And, Or, Not

An advanced search can be constructed by combining several search clauses with the Boolean operators 'And', 'Or' and 'Not'. When a Boolean operator is applied to two clauses (or one in the case of 'Not') the result is a single clause. This resultant clause can also be the subject of another Boolean operation.

The sequence in which these clauses and operators must be declared in the search object is known as Reverse Polish Notation. Clauses (or 'operands') are declared first, and then an Operator is declared. This operates on the last two declared clauses (or the last one for a 'Not' operation). The

clauses affected by the operation are replaced by a single clause representing the Boolean combination.

For example, consider the following sequence of declarations:

```
Clause: A

Clause: B

Operator: Not

Operator: And
```

This results in the logical proposition: 'A and (not B)'.

The following sample code uses RecordSearch object methods:

In Visual Basic

```
objSearch.AddTrayClause(ttWorkTray)

objSearch.AddDateCreatedClause(Date, Date)

objSearch.AddCaveatClause("Medical in Confidence")

objSearch.Not

objSearch.And

objSearch.Or

objSearch.AddLocationClause(objAdminLoc, ltCurrent)

objSearch.And
```

In C#

```
objSearch.AddTrayClause(ttTrayType.ttWorktray);

DateTime dateFrom = new DateTime(2001,1,1);

DateTime dateTo = new DateTime(2002,1,1);

objSearch.AddDateCreatedClause(dateFrom, dateTo);

objSearch.AddCaveatClause("Medical in Confidence");

objSearch.Not();

objSearch.And();

objSearch.Or();

TRIMSDK.Location objAdminLoc = db.GetLocation("Administration");

objSearch.AddLocationClause(objAdminLoc, ltSearchLocationType.ltCurrent, true);

objSearch.And();
```

This results in the search: "(Records in my Worktray or (created today and without the Caveat Ministerial in Confidence)) and currently located in Administration unit".

User Selected Search Criteria

In many cases the programmer will not know the details of the search criteria and instead will delegate the search criteria to the user. To do this, you can call the RecordSearch object's EditQueryUI method. This will display the TRIM Search dialog to the user and update the object's search criteria according to their selections.

You can pre-populate the search criteria by calling a search method before calling the EditQueryUI method. If you specify multiple search methods prior to calling it, the Advanced Search dialog will be displayed.

In Visual Basic

```
Set objSearch = objTRIM.NewRecordSearch

Call objSearch.AddTitleWordClause("Press")

Call objSearch.AddDateRegisteredClause((Date – 1), Date)

Call objSearch.And

If Not objSearch.EditQueryUI(hWnd) Then

Exit Sub  ' (Search dialog cancelled)

End If
```

In C#

```
TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();

objSearch.AddTitleWordClause("Press");

DateTime yesterday = DateTime.Today.AddDays(-1);

DateTime today = DateTime.Today;

objSearch.AddDateRegisteredClause(yesterday, today);

objSearch.And();

int hWnd = Handle.ToInt32();

if (! objSearch.EditQueryUI(hWnd))

{

return;  // (Search dialog cancelled)

}
```

Applying Filters

An optional step in searching for records is to filter the returned records on the basis of Record Type, disposition, class and finalized status. The default is to include all records that meet the criteria, regardless of these categories. To apply filtering, there are methods on the RecordSearch object prefixed with 'Filter…'

In Visual Basic

```
With objSearch
        .AddTitleWordClause("manatee")
        .FilterClass(rcReference)  ' include only Reference class
        .FilterDisposition(rdDestroyed, False) ' include all except Destroyed
        .FilterTypes(colMyTypes)  ' include Types matching this collection
End With
```

In C#

```
objSearch.AddTitleWordClause("manatee");
// include only Reference class
objSearch.FilterClass(rcRecordClass.rcReference,true);
// include all except Destroyed
objSearch.FilterDisposition(rdRecordDisp.rdDestroyed, false);
// include Types matching this collection
objSearch.FilterRecordTypes(colMyTypes);
```

Sorting

Another optional step when constructing a record search is to define the sort order for the search results.

The Sort method allows you to specify up to three different sort criteria, and whether to sort in ascending (the default) or descending order for each.

The following sample code sorts the results by ascending Priority, then Record Type, then descending Date Due.

In Visual Basic

```
Call objSearch.Sort(rsPiority,,rsRecordType,,rsDateDue, True)
```

In C#

```
objSearch.Sort
(rsRecordSortFields.rsPriority,false,rsRecordSortFields.rsRecordType,false,rsRecordS
ortFields.rsDateDue, true);
```

Displaying Results

Once the search criteria, filters and sort order have been specified, you can retrieve the records that match the criteria. These records can either be processed sequentially in code (see Processing Results Sequentially) or they can be copied to a record collection for reporting or displaying to the user.

To copy the results to a Records collection, you must call the GetRecords method.

In Visual Basic

```
Dim objSearch As RecordSearch

Dim colResults As Records

Set objSearch = objTRIM.NewRecordSearch

Call objSearch.AddTitleWordClause("water")

Set colResults = objSearch.GetRecords

Call colResults.DisplayUI(hWnd) ' browse the results
```

In C#

```
TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();

objSearch.AddTitleWordClause("water");

TRIMSDK.Records colResults = objSearch.GetRecords();

int hWnd = Handle.ToInt32();

colResults.DisplayUI(hWnd); // browse the results
```

When the results have been copied to a Records collection, you have several options for displaying records, including allowing the user to select one record (ChooseOneUI), to select multiple records (ChooseManyUI) or simply to browse the results for viewing (DisplayUI).

Processing Results Sequentially

If there is no need to display the search results or to handle them as a collection of records, they can be retrieved one at a time by repeatedly calling the Next method. This returns a single Record object each time it is called (returning a null object when there are no more records to return)

The following sample code performs a search then adds up the values in a User Defined Field called 'Actual Cost', subtotalled by month based on the date the record was created.

In Visual Basic

```
Dim sCosts(12) As Single
Dim iMonth As Integer
' Get the user-defined field "Actual Cost"
Dim objCost As FieldDefinition
Set objCost = objTRIM.GetFieldDefinition("Actual Cost")
' Create the search
Set objSearch = objTRIM.NewRecordSearch
Call objSearch.AddTitleWordClause("Project Cost Report")
' Process the results in a loop
Set objRecord = objSearch.Next
Do Until objRecord Is Nothing
iMonth = Month(objRecord.DateCreated)
sCosts(iMonth) = sCosts(iMonth) + objRecord.GetUserField(objCost)
Set objRecord = objSearch.GetNext
Loop
```

In C#

```
double[] sCosts = new double[12];

int iMonth;

// Get the user-defined field “Actual Cost”

TRIMSDK.FieldDefinition objCost = db.GetFieldDefinition("Actual Cost");

// Create the search

TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();

objSearch.AddTitleWordClause("Project Cost Report");

// Process the results in a loop

TRIMSDK.Record objRecord = objSearch.Next();

while (objRecord != null)

{

        iMonth = objRecord.DateCreated.Month;

        double cost = Convert.ToDouble(objRecord.GetUserField
(objCost,TRIMSDK.sdStringDisplayType.sdDefault))

        sCosts[iMonth] = sCosts[iMonth] + cost;

        objRecord = objSearch.GetNext();

}
```

Simple Record Search

The following sample code adds a TitleWord clause to the Record Search object to find a specified indexed Word.

In Visual Basic

```
// Assumes TRIMDatabase is a valid TRIMSDK Database

// Instantiate a new TRIM record search object

Set RecordSearch = TRIMDatabase.NewRecordSearch

If Not RecordSearch.AddTitleWordClause(txtLookFor.Text) Then

       MsgBox "Add Title Word Clause error " & RecordSearch.ErrorMessage,
vbExclamation

       Exit Sub

End If

// Fill the Records Collection from the Search object

Set RecordResults = RecordSearch.GetRecords

// Instantiate a record by choosing it from the collection

Set RecordItem = RecordResults.ChooseOneUI(hWnd)

If RecordItem Is Nothing Then

         Debug.Print "User cancelled!"

Else

         Debug.Print RecordItem.Number & " - " & RecordItem.Title

End If
```

In C#

```
// Assumes TRIMDatabase is a valid TRIMSDK Database

// Instantiate a new TRIM record search object

TRIMSDK.RecordSearch recordSearch = db.NewRecordSearch();

if (! recordSearch.AddTitleWordClause("title"))

{

       MessageBox.Show( "Add Title Word Clause error " + recordSearch.ErrorMessage,
"", MessageBoxButtons.OK,MessageBoxIcon.Exclamation);

       return;

}

// Fill the Records Collection from the Search object

TRIMSDK.Records recordResults = recordSearch.GetRecords();

// Instantiate a record by choosing it from the collection

int hWnd = Handle.ToInt32();

TRIMSDK.Record recordItem = recordResults.ChooseOneUI(hWnd);

if (recordItem == null)

{

       Console.WriteLine( "User cancelled!");

}

else

{

       Console.WriteLine( recordItem.Number + " - " + recordItem.Title);

}
```

Boolean ‘Or’ Record Search

The following sample code performs a Record search - Adding two TitleWord clauses with an Or to the Record Search object in order to find records with title words of txtSearch1.Text or txtSearch2.Text.

In Visual Basic

```
' Assumes TRIMDatabase is a valid TRIMSDK Database

' Instantiate a new TRIM record search object

Set p_RecordSearch = TRIMDatabase.NewRecordSearch

If Not p_RecordSearch.AddTitleWordClause(txtSearch1.Text) Then

       MsgBox "Add Title Word Clause error " & p_RecordSearch.ErrorMessage,
vbExclamation

       Exit Sub

End If

If Not p_RecordSearch.AddTitleWordClause(txtSearch2.Text) Then

       MsgBox "Add Title Word Clause error " & p_RecordSearch.ErrorMessage,
vbExclamation

       Exit Sub

End If

If Not p_RecordSearch.Or Then

       MsgBox "Adding Boolean 'OR' failed " & p_RecordSearch.ErrorMessage,
vbExclamation

       Exit Sub

End If

' Fill the Records Collection from the Search object

Set p_RecordResults = p_RecordSearch.GetRecords

' Instantiate a record by choosing it from the collection

Set p_RecordItem = p_RecordResults.ChooseOneUI(hWnd)

If p_RecordItem Is Nothing Then

       Debug.Print "User cancelled!"

Else

       Debug.Print p_RecordItem.Number & " - " & p_RecordItem.Title

End If
```

In C#

```
// Assumes TRIMDatabase is a valid TRIMSDK Database

// Instantiate a new TRIM record search object

TRIMSDK.RecordSearch recordSearch = db.NewRecordSearch();

if (! recordSearch.AddTitleWordClause("txtSearch1.Text"))

{

       MessageBox.Show("Add Title Word Clause error " + recordSearch.ErrorMessage,
"", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

       return;

}

if (! recordSearch.AddTitleWordClause("txtSearch2.Text"))

{

       MessageBox.Show( "Add Title Word Clause error " + recordSearch.ErrorMessage,
"", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

       return;

}

if (! recordSearch.Or())

{

      MessageBox.Show( "Adding Boolean 'OR' failed " + recordSearch.ErrorMessage,
"", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

      return;

}

// Fill the Records Collection from the Search object

TRIMSDK.Records recordResults = recordSearch.GetRecords();

// Instantiate a record by choosing it from the collection

TRIMSDK.Record recordItem =     recordResults.ChooseOneUI(Handle.ToInt32());

if (recordItem == null)

{

      Console.WriteLine( "User cancelled!");

}

else

{

      Console.WriteLine( recordItem.Number + " - " + recordItem.Title);

}
```

Saved Search

The following sample code creates a Saved Search (Saving the record search object).

In Visual Basic

```
' Display the properties of a RecordSearch object
' returns True if the user presses OK
If p_RecordSearch.PropertiesUI(hWnd) Then
        If p_RecordSearch.Verify(True) Then
               ' If no errors or warnings, Save the Record Search
                  p_RecordSearch.Save
                  MsgBox "Saved Search created - " & p_RecordSearch.Name,
vbInformation
        Else
       ' Display Errors
        MsgBox "Record Search Verify failed: " & p_RecordSearch.ErrorMessage,
vbExclamation
        End If
End If
```

In C#

```
TRIMSDK.RecordSearch recordSearch = db.NewRecordSearch();

int hWnd = Handle.ToInt32();

// Display the properties of a RecordSearch object

// returns True if the user presses OK

if (recordSearch.PropertiesUI(hWnd))

{

       if (recordSearch.Verify(true))

       {

            // If no errors or warnings, Save the Record Search

             recordSearch.Save();

             MessageBox.Show( "Saved Search created - " + recordSearch.Name, "",
MessageBoxButtons.OK, MessageBoxIcon.Information);

       }

       else

       {

           // Display Errors

            MessageBox.Show( "Record Search Verify failed: " +
recordSearch.ErrorMessage, "", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

       }
```

Creating a Container File

This scenario describes the general processes for using the SDK to create a record of a generic Record Type we are calling a 'Container File'. In this and the next scenario (Creating a Document) we are assuming that the reader is familiar with the concept of Record Types. These are described in Content Manager Help – Administrator Guide – Record Types.

While it is up to the Administrator of each TRIM implementation to determine the Record Types to be used, it is typical to follow a standard records management practise of having at least two Record Types, one representing Container Files (or Folders) and one representing Documents (the actual names used for the Record Types may of course vary). Container Files are usually created and maintained by specialist records managers, as it is generally at this level that Classification systems, Retention Schedules, Security, Thesaurus Terms (keywords), controlled titling and other records management metadata are applied. Documents, on the other hand, are usually created by end-users, and require little specific metadata other than the identification of the appropriate Container File to which the Document belongs, as all other metadata and context is inherited from the Container.

The general steps for creating a new Container File record are as follows:

1. Instantiate the appropriate Record Type object
2. Instantiate a new Record object of this Type
3. Identify the Classification or Keywords for titling the record (optional)
4. Set the free text title
5. Assign Security Levels and Caveats (optional)
6. Relate the record to associated Locations (optional)
7. Relate to other records (optional)
8. Assign other metadata or User Defined Fields (optional)
9. Assign a record identifier
10. Save the Record object.

Creating a Record of a given Type

When creating any record, the Record Type for the new record must be identified. This can be done programmatically if the Record Type's URI or Name is known at design-time, or the choice may be given to the user at run-time. In either case, the end result is to instantiate an existing Record Type object (using the GetRecordType method), and to pass this object to the Database object's NewRecord method.

In Visual Basic

```
' Create a new Case File record

Set objRecType = objTRIM.GetRecordType("Case File")

Set objRecord = objTRIM.NewRecord(objRecType)
```

In C#

```
//Create a new Case File record

TRIMSDK.RecordType objRecType = db.GetRecordType("Case File");

TRIMSDK.Record objRecord = db.NewRecord(objRecType);
```

> **NOTE:** It is possible to create new Record Types using the COM SDK; however, this is not recommended as this is generally an Administrator's function only).

Controlled and Free Text Titling

Titles for Container Files are often subject to controlled vocabulary or Classification structures such as a Thesaurus or Classification (file) plan, which give records managers greater control over file creation, retrieval and Retention. Even when such controlled titling is used, each file will typically also have a 'free text' title part. The titling method used is determined by the Record Type, and is usually set by the TRIM Administrator. Thus a record with Classification titling may have a title such as: "Insurance – Property – Storm damage to Mackay information center", where the first two terms are generated from a predefined hierarchical Classification structure and the remaining part of the title is

'free text' describing the specifics of the file. The generated title terms are determined by the Classification codes, usually defined as a numerical sequence such as "610/600/". The free text title is set via the TypedTitle property.

In Visual Basic

```
If objRecordType.TitlingMethod = tmClassification Then
      ' Assign classification of 610/600/ = Insurance – Property
       objRecord.Classification = objTRIM.GetClassification("610/600/")
End If
objRecord.TypedTitle = "Storm damage to Mackay information center"
```

In C#

```
if (objRecordType.TitlingMethod == tmTitlingMethods.tmClassification)
{
       // Assign classification of 610/600/ = Insurance - Property
        objRecord.Classification = db.GetClassification("610/600/");
}
objRecord.TypedTitle = "Storm damage to Mackay information center";
```

Similarly, Thesaurus (or Keyword) titling allows a file to be titled using either a choice of individual keywords from a controlled list or a specific 'branch' of related terms according to a hierarchical structure (similar to a record plan or Classification). A Thesaurus-titled file might have a name such as "Administration – Finance – Donations – Bequest from the estate of Lady Marchcroft".

In Visual Basic

```
objRecord.GeneratedTitle = "Administration - Finance - Donations"
objRecord.TypedTitle = "Bequest from the estate of Lady Marchcroft"
```

In C#

```
objRecord.GeneratedTitle = "Administration - Finance - Donations";
objRecord.TypedTitle = "Bequest from the estate of Lady Marchcroft";
```

Security Levels and Caveats

The security profile of an individual TRIM record is governed by three security controls: a Security Level, a set of zero or more Caveats, and Access Control.

(See *Content Manager Help File* – Administrator Guide – Ch1 - Security).

Access Control is discussed in the next section.

Security Levels and Caveats determine the access that a TRIM user has to the metadata of a record. These security specifications are usually applied to Record Types (and inherited by records of each type when they are created) but can be set explicitly on individual records. Every user has a maximum Security Level and zero or more Caveats – in order to access a particular record, the user must have the same or a higher Security Level and must have all the Caveats associated with the record.

Assigning Security Levels and Caveats to a record via the SDK is straightforward. Both the SecurityLevel object and the SecurityCaveat object can be instantiated by full name or by abbreviation. The instantiated SecurityLevel object is assigned to the record's SecLevel property. Each instantiated SecurityCaveat object can be passed to the record's AddCaveat method.

In Visual Basic

```
Dim objCav as SecurityCaveat

Dim objSec as SecurityLevel

' Assign "Confidential" level

Set objSec = objTRIM.GetSecurityLevel("Confidential")

Set objRecord.SecLevel = objSec

' Assign "Research Projects" Caveat

Set objCav = objTRIM.GetSecurityCaveat("Research Projects")

Call objRecord.AddCaveat(objCav)

' Assign "Staff in Confidence" Caveat

Call objRecord.AddCaveat(objTRIM.GetSecurityCaveat("Staff in Confidence"))
```

In C#

```
// Assign "Confidential" level

TRIMSDK.SecurityLevel objSec = db.GetSecurityLevel("Confidential");

objRecord.SecLevel = objSec;

// Assign "Research Projects" Caveat

TRIMSDK.SecurityCaveat objCav = db.GetSecurityCaveat("Research Projects");

objRecord.AddCaveat(objCav);

// Assign "Staff in Confidence" Caveat

objRecord.AddCaveat(db.GetSecurityCaveat("Staff in Confidence"));
```

> **NOTE:** That it is also possible to assign a string value of comma-separated Security Level and Caveat names (not abbreviations) to the Record object's Security property. If the string can be completely parsed into legal security values, they will be assigned to the record. The following sample code produces the same result as the sample above.

In Visual Basic

```
' Assign security level and two Caveats

objRecord.Security = "Confidential, Research Projects, Staff in Confidence"
```

In C#

```
// Assign security level and two Caveats

objRecord.Security = "Confidential, Research Projects, Staff in Confidence";
```

Access Control

In addition to Security Levels and Caveats, Access Control provides fine-grained control over different methods of access to a record and its electronic attachment. (See *Content Manager Help File* – Administrator Guide – Ch 1- Security – Access Control).

Access Control associates individual users or groups of users with specific actions allowed for a record. The actions are:

- reading metadata
- updating metadata
- viewing the electronic object
- updating the electronic object
- deleting the record
- changing Access Control details

Each action can be granted access as follows:

- Public (all users)
- Private (only one user)
- Inherited (same access as the Container record)
- Ad hoc (a set of named locations)

The default for a record that has no Access Control specified is that all users can perform all actions (subject to Security Levels and Caveats).

Access Control is normally applied to individual Container records, and may be inherited by contained records or explicitly set for each contained record.

The SetAccessControlDetails method of the Record object is used to add specifications of the Access Control for the record. This method requires that you specify one of the six actions listed above and the access level (including the locations, if private or ad-hoc).

The following sample code grants these rights:

- Public access to view the metadata
- Inherited access to update the metadata

Only the Records Manager can delete the record.

> **NOTE:** That the connected user must have 'Modify Access Control' permission for this code to work.

In Visual Basic

```
Call objRecord.SetAccessControlDetails(dxViewRecord, asPublic)

Call objRecord.SetAccessControlDetails(dxUpdateMetadata, asInherited)

Call objRecord.SetAccessControlDetails(dxDeleteRecord, asPrivate,
objTRIM.GetLocation("Records Manager"))
```

In C#:

```
objRecord.SetAccessControlDetails(dxRecordAccess.dxViewRecord,
asAccessControlSettings.asPublic,null);

objRecord.SetAccessControlDetails(dxRecordAccess.dxUpdateMetadata,
asAccessControlSettings.asInherited,null);

objRecord.SetAccessControlDetails(dxRecordAccess.dxDeleteRecord,
asAccessControlSettings.asPrivate, db.GetLocation("Records Manager"));
```

Relationships

The context of a document in TRIM is generally provided by the Container file in which it is logically enclosed. To provide useful context for a Container file record, you can use various techniques such as a Classification system. You can also provide context by creating relationships with other records in the Database. TRIM defines some standard relationship types, but you can also create custom relationship definitions. Apart from the generic type of "related", all relationship types in TRIM are transitive, meaning that the relationship has a subject and an object (for example, the transitive relationship "A supersedes B" is not the same as "B supersedes A").

In the SDK, you use the Record object's AttachRelationship method to relate another record to the current record. The record on which the method is being called is the subject of the relationship, and the other record (passed as an argument to the method) is the object. The relationship type is determined by passing a value of the rrRecordRelationship enumeration.

The following sample code creates a relationship of "Record A supersedes Record B".

In Visual Basic

```
objRecordA.AttachRelationship(objRecordB, rrDoesSupersede)
```

In C#

```
objRecordA.AttachRelationship(objRecordB, rrRecordRelationship.rrDoesSupersede);
```

Record Locations

Defining relationships between a Container file and location objects (people and places) provides additional and useful context for the record.

Unlike record relationships, which can be user defined, you can only use TRIM's predefined standard relationship types for record locations (and for contacts, see Record Contacts).

Record Locations represent actual (in the case of paper and other physical records) or logical (in the case of electronic records) places where a record resides. Every record in TRIM has a property representing it's Current Location (where the record is now) and another for it's Home Location (where the record should normally be or where it is to be returned). There is also a property for Owner Location – the exact meaning of this can vary according to the practises of each TRIM implementation, but normally represents the person or body that is responsible for the record. The Home and Owner location of a record are typically derived from the default values for each Record Type, but all record location properties can be set on creation of a new record or modified later.

The Record object has methods for setting or changing the value of these location properties, which allow the option of specifying the date & time of the change of location (the default is the current time).

The following sample code sets the record's Home location to the unit called "Administration", and the Current location to the connected user.

In Visual Basic

```
objRecord.SetHomeLocation(objTRIM.GetLocation("Administration"))

objRecord.SetCurrentLocation(objTRIM.CurrentUser)
```

In C#

```
objRecord.SetHomeLocation(db.GetLocation("Administration"));

objRecord.SetCurrentLocation(db.CurrentUser,DateTime.Now);
```

Record Contacts

Unlike record locations (see Record Locations), which tend to be internal units, Record Contacts are more commonly people or organisations that have a direct association with the record, and may be internal or external to the organisation. Using the AttachContact method, TRIM allows each contact to be specifically identified as an Author, Addressee, Representative or Client. Other contact relationship types must use the generic type of 'Other'.

The following sample code sets the record's Representative (and primary contact) to be the connected user, and the Client to be the organisation called "My Organization".

In Visual Basic

```
objRecord.AttachContact(objTRIM.CurrentUser, ctRepresentative, True)

objRecord.AttachContact(objTRIM.GetLocation("My Organization"), ctClient)

' ctClient = Client
```

In C#

```
objRecord.AttachContact(db.CurrentUser, ctContactType.ctRepresentative,
true,DateTime.Now);

objRecord.AttachContact(db.GetLocation("My Organization"),
ctContactType.ctClient,false,DateTime.Now);

// ctClient = Client
```

General Code Examples

The following sample code demonstrates many of the features described above. The code will work with the Demonstration Database.

In Visual Basic

```
Dim objTRIM As New Database

Dim objRecord As Record

Dim objRecordB As Record

' Create a new File Folder record

Set objRecordType = objTRIM.GetRecordType("Research Project File")

Set objRecord = objTRIM.NewRecord(objRecordType)

With objRecord

' Set keyword title and free text title

         .GeneratedTitle = "Administration - Finance - Donations"

         .TypedTitle = "Bequest from the estate of Lady Marchcroft"

        ' Relate to the superseded record

         Set objRecordB = objTRIM.GetRecord("76/915")

         Call .AttachRelationship(objRecordB, rrDoesSupersede)

        ' Assign "Confidential" security level

         .SecLevel = objTRIM.GetSecurityLevel("Confidential")

        ' Add "Research Projects" Caveat

         Call .AddCaveat(objTRIM.GetSecurityCaveat("Research Projects"))

        ' Access Control - only this user can update

         Call .SetAccessControlDetails(dxUpdateMetadata, asPrivate,
objTRIM.CurrentUser)

        ' Locations

         Call .SetHomeLocation(objTRIM.GetLocation("Administration"))

         Call .SetCurrentLocation(objTRIM.CurrentUser)

        ' Contacts

         Call .AttachContact(objTRIM.CurrentUser, ctAuthor, True)

         Call .AttachContact(objTRIM.GetLocation("Bay Books"), ctClient)

        ' Verify and Save

         If Not .Verify Then
```

```
            MsgBox .ErrorMessage
            Else
            .Save
            End If
    End With
```

In C#

```
TRIMSDK.Database db = new TRIMSDK.Database();
// Create a new File Folder record
TRIMSDK.RecordType objRecordType = db.GetRecordType("Research Project File");
TRIMSDK.Record objRecord = db.NewRecord(objRecordType);
// Set keyword title and free text title
objRecord.GeneratedTitle = "Administration - Finance - Donations";
objRecord.TypedTitle = "Bequest from the estate of Lady Marchcroft";
// Relate to the superseded record
TRIMSDK.Record objRecordB = db.GetRecord("76/915");
objRecord.AttachRelationship(objRecordB, rrRecordRelationship.rrDoesSupersede);
// Assign "Confidential" security level
objRecord.SecLevel = db.GetSecurityLevel("Confidential");
// Add "Research Projects" Caveat
objRecord.AddCaveat(db.GetSecurityCaveat("Research Projects"));
// Access Control - only this user can update
objRecord.SetAccessControlDetails(dxRecordAccess.dxUpdateMetadata,
asAccessControlSettings.asPrivate, db.CurrentUser);
// Locations
objRecord.SetHomeLocation(db.GetLocation("Administration"));
objRecord.SetCurrentLocation(db.CurrentUser,DateTime.Now);
// Contacts
objRecord.AttachContact(db.CurrentUser, ctContactType.ctAuthor, true,DateTime.Now);
objRecord.AttachContact(db.GetLocation("Bay Books"),
ctContactType.ctClient,false,DateTime.Now);
// Verify and Save
if (! objRecord.Verify(false))
{
      MessageBox.Show (objRecord.ErrorMessage);
}
else
{
      objRecord.Save();
```

```
}
```

Creating a Document

This scenario describes the general processes for using the SDK to create a record of a generic Record Type we are calling a 'Document'.

(See Searching for Records - Creating a Container File).

While Container Files are usually created and maintained by specialist records managers, Documents, on the other hand, are usually created by end-users, and require little specific metadata other than the identification of the appropriate Container File to which the Document belongs, as most other metadata and context is inherited from the Container. A Document record usually consists of an electronic object (the source document, image or other file), a unique identifier (which may be automatically generated by TRIM), a record title and any other metadata required to profile and index the record, and a pointer to the Container File from which the document derives its context.

The general steps for creating a new Document record are as follows:

1. Instantiate the appropriate Record Type object
2. Instantiate a new Record object of this Type
3. Identify the Container File for the document
4. Set the free text title
5. Attach an Electronic file
6. Assign the record's Author or other contacts (optional)
7. Set Access Control to the elecronic document (optional)
8. Assign other metadata or User Defined Fields (optional)
9. Save the Record object.

Titling and Numbering

Titling for documents is generally straightforward – free text titling is the norm, and the title simply needs to succinctly describe the document or record. Record numbers may be assigned explicitly or they may be automatically generated – this is configured on the Record Type properties. If the number is explicitly assigned, the number (in expanded format) must be assigned to the LongNumber property (it must be unique or the record will not be saved).

In Visual Basic

```
objRecord.Title = "Letter from executor regarding disbursements of Lady Marchcroft's bequest"

objRecord.LongNumber = "XK/008934"
```

In C#

```
objRecord.Title = "Letter from executor regarding disbursements of Lady Marchcroft’s bequest";

objRecord.LongNumber = "XK/008936";
```

Assigning to a Container

Although it is not compulsory, it is most common that an electronic record is logically assigned to a Container file that represents the subject matter, case, client file or other contextual grouping relevant to the document.

To assign a record to a Container, the existing Container record must be instantiated (by Id or URI) and then passed as an argument to the (contained) record object's SetContainer method. The method includes a parameter for specifying whether the record is also 'enclosed in' the Container, i.e. that the current location should reflect that it is with the Container.

In Visual Basic

```
Dim objContainer As Record

Set objContainer = objTRIM.GetRecord("76/915")

objRecord.SetContainer(objContainer, True)
```

In C#

```
TRIMSDK.Record objContainer = db.GetRecord("76/915");

objRecord.SetContainer(objContainer, true);
```

Attaching an Electronic Document

Document records can represent physical paper documents, but mostly they will include an electronic attachment, whether this is a word-processing document, scanned image or other type of file.

To attach an electronic document to a record, the file name and path must be used to instantiate an InputDocument object. This object is then passed as an argument to the record object's SetDocument method. The method includes parameters for specifying whether this should replace any existing document (or be added as a new revision), whether it should be marked as checked out to the current user, and any comments to be added to the record's Notes field.

In Visual Basic

```
Dim objDoc As New InputDocument

Call objDoc.SetAsFile("C:\myDocs\ThisFile.doc")

Call objRecord.SetDocument(objDoc, False, False, "Created via SDK")
```

In C#

```
TRIMSDK.InputDocument objDoc = new InputDocument();

// note that in C# the \ character is an escape symbol,

// unless the string is preceded by an @.

objDoc.SetAsFile(@"C:\myDocs\ThisFile.doc");

objRecord.SetDocument(objDoc, false, false, "Created via SDK");
```

Alternatively, if the file to be attached is not known until run-time, you can call the SetDocumentUI method, which will display a dialog for the user to select the file.

In Visual Basic

```
If Not objRecord.SetDocumentUI(hWnd, "TheDefault.doc", "Attach Document", False)
Then

       Msgbox "Action cancelled."

       Exit Sub

End If
```

In C#

```
int hWnd = Handle.ToInt32();

if (! objRecord.SetDocumentUI(hWnd, "", "Attach Document", false))

{

      MessageBox.Show("Action cancelled.");

}
```

Document Author

Record Contacts are TRIM location objects commonly representing people or organisations that have a direct association with the record. The most common type of Contact to be specified for an electronic document is the Author. Although the AttachContact method can be used for this and other contact types, a shortcut is provided through the AuthorLoc property.

The following sample code sets the document's Author to be the connected user.

In Visual Basic

```
objRecord.AuthorLoc = objTRIM.CurrentUser
```

```
In C#objRecord.AuthorLoc = db.CurrentUser;
```

Access Control for Documents

For more information on this subject, see Searching for Records - Access Control.

The SetAccessControlDetails method of the Record object is used to add specifications of the Access Control for the record. This method requires that you specify one of the six actions listed above and the access level (including the locations, if private or ad-hoc). For Document records, the typical action is to assign View and Update rights to the electronic document.

The following sample code grants the following:

- Private access to the connected user for updating the electronic document
- Public access to view the Document.

In Visual Basic

```
Call objRecord.SetAccessControlDetails(dxUpdateDocument, asPrivate,
objTRIM.CurrentUser)
```

```
Call objRecord.SetAccessControlDetails(dxViewDocument, asPublic)
```

In C#

```
objRecord.SetAccessControlDetails(dxRecordAccess.dxUpdateDocument,
asAccessControlSettings.asPrivate, db.CurrentUser);
```

```
objRecord.SetAccessControlDetails(dxRecordAccess.dxViewDocument,
asAccessControlSettings.asPublic,null);
```

Setting User-Defined Fields

Any type of record can have any number of User Defined Fields associated with it. (For background information on User Defined Fields, see Object Properties - The FieldDefinition Object)

To assign values to User Defined Fields on a record, you must instantiate a FieldDefinition object representing the User Defined Field, and pass this and a Variant containing the data value to the Record object's SetUserField method.

The following sample code assumes that a User Defined String Field called "Job Code" has been created in TRIM. It assigns a value of "D0933" to this field on the current record.

In Visual Basic

```
Call objRecord.SetUserField(objTRIM.GetFieldDefinition("Job Code"), "D0933")
```

```
In C#objRecord.SetUserField(db.GetFieldDefinition("Job Code"), "D0933");
```

Creating a Record with user input

In Visual Basic

```
' Modular level (m_)
Private m_TRIMDatabase As TRIMSDK.Database
' Procedural level variables (p_)
Dim p_RecordTypes  As TRIMSDK.RecordTypes
Dim p_RecordType   As TRIMSDK.RecordType
Dim p_NewRecord    As TRIMSDK.Record
' Instantiate a collection of Record Types.
Set p_RecordTypes = m_TRIMDatabase.MakeRecordTypes
' Fill the collection with all Record Types, before filtering
p_RecordTypes.SelectAll
' Instantiate a Record Type by choosing it from the collection
Set p_RecordType = p_RecordTypes.ChooseOneUI(hWnd)
If p_RecordType Is Nothing Then
       Debug.Print "User pressed Cancel"
       Exit Sub
End If
' Instantiate a new Record of the Record Type passed in.
Set p_NewRecord = m_TRIMDatabase.NewRecord(p_RecordType)
' Display the properties of new Record
' Returns True if the user selects OK.
       If p_NewRecord.PropertiesUI(hWnd) Then
       If p_NewRecord.Verify Then
               p_NewRecord.Save
               MsgBox "Created a new record - " & p_NewRecord.Number
       Else
               MsgBox "Error saving new Record properties " & _p_
NewRecord.ErrorMessage, vbExclamation
       End If
End If
' Clean Up
```

```
Set p_RecordTypes = Nothing
Set p_RecordType = Nothing
Set p_NewRecord = Nothing
```

In C#

```
private TRIMSDK.Database db = new TRIMSDK.Database();
// Instantiate a collection of Record Types.
TRIMSDK.RecordTypes recordTypes = db.MakeRecordTypes();
// Fill the collection with all Record Types, before filtering
recordTypes.SelectAll();
// Instantiate a Record Type by choosing it from the collection
int hWnd = Handle.ToInt32();
TRIMSDK.RecordType recordType = recordTypes.ChooseOneUI(hWnd);
if (recordType == null)
{
	Console.WriteLine( "User pressed Cancel");
	return;
}
// Instantiate a new Record of the Record Type passed in.
TRIMSDK.Record newRecord = db.NewRecord(recordType);
// Display the properties of new Record
// Returns True if the user selects OK.
if (newRecord.PropertiesUI(hWnd))
{
	if (newRecord.Verify(false))
	{
		newRecord.Save();
		MessageBox.Show( "Created a new record - " + newRecord.Number);
	 }
	 else
	 {
		MessageBox.Show( "Error saving new Record properties " +
newRecord.ErrorMessage, "", MessageBoxButtons.OK,MessageBoxIcon.Exclamation);
	 }
}
// Clean Up
recordTypes = null;
```

```
recordType = null;
newRecord = null;
```

Creating a Record with no user input

In Visual Basic

```
' Modular level (m_)
Private m_TRIMDatabase As TRIMSDK.Database
' Procedural level variables (p_)
Dim p_RecordTypes  As TRIMSDK.RecordTypes
Dim p_RecordType   As TRIMSDK.RecordType
Dim p_NewRecord    As TRIMSDK.Record
Set m_TRIMDatabase = New TRIMSDK.Database
On Error GoTo err_handler
'// Instantiate a Record Type from its name or Uri
Set p_RecordType = m_TRIMDatabase.GetRecordType("Research Project File")
If p_RecordType Is Nothing Then
      '// Name or Uri did not uniquely identify a record type.
       Debug.Print "Error instantiating Record Type."
       Exit Sub
End If
Set p_HomeLocation = m_TRIMDatabase.GetLocation("Llewellyn, Brian (Professor) OBE")
If p_HomeLocation Is Nothing Then
      '// Name or Uri did not uniquely identify a TRIM Location.
       Debug.Print "Error instantiating Location: " & p_RecordType.ErrorMessage
       Exit Sub
End If
'// Instantiate a new Record of the Record Type passed in.
Set p_NewRecord = m_TRIMDatabase.NewRecord(p_RecordType)
'// Complete all of the new record's properties.
With p_NewRecord
      '// An error is raised if any of these properties fail.
      '// Thesaurus titling
      .GeneratedTitle = "ADMINISTRATION - FINANCE - LEASES AND RENTAL AGREEMENTS -
SUPPLIER [Larger than Life Ventures]"
' p_Keyword.Name
```

```
        '// Free text titling
         .TypedTitle = "New Record Title"
        '// Record's Home location
         .SetHomeLocation p_HomeLocation
         If p_NewRecord.Verify Then
                p_NewRecord.Save
                Debug.Print "Created a new record - " & p_NewRecord.Number
         Else
                MsgBox "Error saving new Record" & p_NewRecord.ErrorMessage,
vbExclamation
         End If
End With
Set p_RecordType = Nothing
Set p_NewRecord = Nothing
Set p_HomeLocation = Nothing
Exit Sub
err_handler:
'// The error message is also populated in the Err object.
MsgBox "Error: " & Err.Description, vbExclamation
Set p_RecordType = Nothing
Set p_NewRecord = Nothing
Set p_HomeLocation = Nothing
```

In C#

```
TRIMSDK.Database db = new TRIMSDK.Database();
try
{
       // Instantiate a Record Type from its name or Uri
        TRIMSDK.RecordType recordType = db.GetRecordType("Research Project File");
        if (recordType == null)
        {
                // Name or Uri did not uniquely identify a record type.
                 Console.WriteLine ("Error instantiating Record Type.");
                 return;
        }
        TRIMSDK.Location homeLocation = db.GetLocation("Llewellyn, Brian (Professor)
OBE");
        if (homeLocation == null)
        {
                // Name or Uri did not uniquely identify a TRIM Location.
                 Console.WriteLine( "Error instantiating Location: " +
recordType.ErrorMessage);
                 return;
         }
        // Instantiate a new Record of the Record Type passed in.
         TRIMSDK.Record newRecord = db.NewRecord(recordType);
         // Complete all of the new record's properties.
         // An error is raised if any of these properties fail.
         // Thesaurus titling
         newRecord.GeneratedTitle = "ADMINISTRATION - FINANCE - LEASES AND RENTAL
AGREEMENTS - SUPPLIER [Larger than Life Ventures]"; //p_Keyword.Name
        // Free text titling
         newRecord.TypedTitle = "New Record Title";
         // Record's Home location
          if (newRecord.Verify(false))
          {
                 newRecord.Save();
```

```
                Console.WriteLine( "Created a new record - " + newRecord.Number);
            }
            else
            {
                MessageBox.Show( "Error saving new Record" +
newRecord.ErrorMessage,"",MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
             }
             recordType = null;
             newRecord = null;
             homeLocation = null;
            return;
             }
             catch(Exception ex)
             {
                // The error message is also populated in the ex object.
                 MessageBox.Show( "Error: " + ex.Message, "", MessageBoxButtons.OK,
MessageBoxIcon.Exclamation);
              }
```

Checking Out a Document

When an electronic document in TRIM needs to be updated, it must first be checked out to a user, to prevent others from attempting to edit the same document. On completion of the changes, it can be check-in to make the updated version available in TRIM.

Locating the Document

Various methods can be used to locate and instantiate the document that is to be checked out. If the unique identifier (Record Number or URI) is known, it can be passed to the Database object's GetRecord method. Alternatively the record can be located by the user, either through an interactive search or by selecting from the contents of a specific Container file.

The following sample code combines some elements of the options described above, by using a Records collection to display the contents of a specific Container file, and instantiating the record that the user selects from a displayed result list.

In Visual Basic

```
Dim objContainer As Record

Dim objDoc As Record

Dim colContents As Records

Set objContainer = objTRIM.GetRecord("96/715")

Set colContents = objTRIM.MakeRecords

Call colContents.SelectContentsOf(objContainer)

Set objDoc = colContents.ChooseOneUI(hWnd)

If Not objDoc.IsElectronic Then

        Exit Sub

End If
```

In C#

```
TRIMSDK.Record objContainer = db.GetRecord("96/715");

TRIMSDK.Records colContents = db.MakeRecords();

colContents.SelectContentsOf(objContainer);

int hWnd = Handle.ToInt32();

TRIMSDK.Record objDoc = colContents.ChooseOneUI(hWnd);

if ( objDoc != null && ! objDoc.IsElectronic)

{

        return;

}
```

Check Out Options

Once the appropriate electronic document has been identified and instantiated, the object can be programmatically checked out to a specific file destination by calling the GetDocument method.

In Visual Basic

```
If objDoc.IsElectronic Then

       Call objDoc.GetDocument("C:\tmp\MyFile.doc", True, "Checked out via SDK")

End If
```

In C#

```
if (objDoc.IsElectronic)
{
	objDoc.GetDocument(@"C:\tmp\MyFile.doc", true, "Checked out via SDK", " ");
}
```

Alternatively, a user can choose a document to check out to their TopDrawer via a dialog by calling TopDrawerDisplayUI on a collection of records.

The following sample code allows a user to select from their list of favorite documents.

In Visual Basic

```
Call colRecords.SelectFavorites
Call colRecords.TopDrawerDisplayUI(hWnd)
```

In C#

```
colRecords.SelectFavorites();
int hWnd = Handle.ToInt32();
colRecords.TopDrawerDisplayUI(hWnd);
```

Checking In a Document

After a document has been edited and it is ready to be returned to TRIM, it must be Checked-in. This can be done manually either through the TRIM or TopDrawer clients (if the document was checked out to TopDrawer). To Check-in a document programmatically, you must use the SetDocument method of the record that has been checked-out. The method provides options for adding notes and specifying whether the latest revision should replace the current one or be stored as a new revision.

In Visual Basic

```
If objDoc.CheckedOutTo.Uri = objTRIM.CurrentUser.Uri Then
	objDoc.SetDocument("C:\tmp\MyFile.doc", True, False, "Checked in via SDK")
End If
```

In C#

```
if (objDoc.CheckedOutTo.Uri == db.CurrentUser.Uri)
{
	TRIMSDK.InputDocument document = new InputDocument();
	document.SetAsFile(@"C:\myDocs\ThisFile.doc");
	objDoc.SetDocument(document, true, false, "Checked in via SDK");
}
```

To check-in a document interactively, you can use the SetDocumentUI method. This will display a TRIM dialog to allow the user to specify the check-in options.

In Visual Basic

```
If Not SetDocumentUI(hWnd, "MyFile.doc", "Check-In", False) Then

       MsgBox "Check-in cancelled"

End If
```

In C#

```
int hWnd = Handle.ToInt32();

if (! objDoc.SetDocumentUI(hWnd, "", "Check-In", false))

{

      MessageBox.Show("Check-in cancelled");

}
```

Working with Locations

The Location object is an encapsulation of all properties and methods associated with Persons, Organizations, Positions and Groups. Locations can be identified by name or by URI, and can be selected on other criteria, such as date of birth, nicknames, or membership of a particular organization, role or group.

Finding a Person by Name

Although the names of non-persons (Units, Positions and Organizations) must be unique, this is not the case for persons (Staff Names & Contacts). However, TRIM allows you to store a 'nickname' for any person, and this can be used as a substitute for a persons name when searching.

To find a particular person by name, you must pass the person's combined name and title to the Database object's GetLocation method.

In Visual Basic

```
Dim objLoc As Location

Set objLoc = objTRIM.GetLocation("Abbott, Peter (Mr)")
```

In C#

```
TRIMSDK.Location objLoc = db.GetLocation("Abbott, Peter (Mr)");
```

Alternatively, you can pass a sub-string of the person's name followed by a wildcard (asterisk) character, as long as the text provided uniquely identifies a location.

In Visual Basic

```
Set objLoc = objTRIM.GetLocation("Abbott, P*")

Set objLoc = objTRIM.GetLocation("Abbott*")

Set objLoc = objTRIM.GetLocation("Abbott, Peter*")
```

In C#

```
objLoc = db.GetLocation("Abbott, P*");

objLoc = db.GetLocation("Abbott*");

objLoc = db.GetLocation("Abbott, Peter*");
```

If the sub-string does not uniquely identify a location (i.e. there are no matches, or there is more than one match) then a null object will be returned.

In Visual Basic

```
Set objLoc = objTRIM.GetLocation("Abb*") ' finds Abbott and Abbey

If objLoc Is Nothing Then Exit Sub
```

In C#

```
objLoc = db.GetLocation("Abb*"); //finds Abbott and Abbey

if (objLoc == null)

{

       return;

}
```

Creating a New Staff Member

To create a new staff member, you must instantiate a new location by calling the NewLocation method on the Database object. You then define the type of the location by assigning a value (in this case lcPerson) to the LocType property. You can then set various properties representing the person's name, contact details such as telephone numbers and addresses, administrative details such as employee ID numbers and so on.

If the new person is to be a TRIM user, then there are login and security details to be provided. You will need to specify the user's network login ID and optionally an expiry date. For the security profile, you are required to either explicitly state the user's Security Level (and optionally any Caveats) and a user category, or if role-based security is used you can specify that the user takes the profile of a predefined group or user.

Relationships such as membership of units or reporting lines are created using the AddRelationship method and passing parameters for the related location and the relationship type.

Addresses (including electronic addresses such as email or URL) are added by calling the New method on the LocAddresses or LocEAddresses collection properties.

In Visual Basic

```
Dim objUnit As Location
Dim objBoss As Location
Dim objPeer As Location
Dim objRole As Location
Dim objSec As SecurityLevel
Dim objEmail As LocEAddress
Dim bRoleSecurity As Boolean
bRoleSecurity = False
Set objRole = objTRIM.GetLocation("Project Manager")
Set objLoc = objTRIM.NewLocation
With objLoc
       .LocType = lcPerson
' Name
       .Surname = "Evans"
       .GivenNames = "David"
       .Initial1 = "D"
       .Initial2 = "W"
       .Honorific = "Mr"
' Personal & Administrative
       .IsWithin = True      ' Internal to the org
       .IdNumber = 793906
       .ReviewDate = Date + 365
       .DateOfBirth = #11/29/1966#
       .PhoneNo = "555 123496"
       .MobileNo = "+44 7939 062736"
       .Notes = "Created via SDK"
' Login details
       .CanLogin = True
       .LoginExpires = Date + (365 * 3)  ' Valid for 3 years
       .LogsInAs = "evans"         ' Network login id
' Security
       If bRoleSecurity Then
```

```
                .UseProfileOf = objRole
        Else
                Set objSec = objTRIM.GetSecurityLevel("Confidential")
                .SecLevel = objSec
                .UserType = utRecordsWorker
        End If
    ' Email address
        Set objEmail = .LocEAddresses.New
        objEmail.EAddressType = etMail
        objEmail.EAddress = "david@gbrmpa.com.au"
        objEmail.Description = "Default business email"
    ' Relationships
        Call .AddRelationship(objRole, lrHasGroups)
        Set objUnit = objTRIM.GetLocation("Administration")
        Call .AddRelationship(objUnit, lrMemberOf, True)
        Set objBoss = objTRIM.GetLocation("Neumann, Ilse*")
        Call .AddRelationship(objBoss, lrBossedBy)
    ' Confirm & Save
        If .Verify(True) Then
               .Save
               MsgBox .FormattedName & " created."
        Else
               MsgBox .ErrorMessage
        End If
    End With
```

In C#

```
bool bRoleSecurity = false;
TRIMSDK.Location objRole = db.GetLocation("Project Manager");
TRIMSDK.Location objLoc = db.NewLocation();
objLoc.LocType = lcLocationType.lcPerson;
// Name
objLoc.Surname = "Evans";
objLoc.GivenNames = "David";
objLoc.Initial1 = "D";
objLoc.Initial2 = "W";
objLoc.Honorific = "Mr";
// Personal & Administrative
objLoc.IsWithin = true; // Internal to the org
objLoc.IdNumber = Convert.ToString(793906);
objLoc.ReviewDate = DateTime.Today.AddYears(1);
DateTime dob = new DateTime(1966,11,29);
objLoc.DateOfBirth = dob;
objLoc.PhoneNo = "555 123496";
objLoc.MobileNo = "+44 7939 062736";
objLoc.Notes = "Created via SDK";
// Login details
objLoc.CanLogin = true;
objLoc.LoginExpires = DateTime.Today.AddYears(3); // Valid for 3 yrs
objLoc.LogsInAs = "evans"; // Network login id
// Security
if (bRoleSecurity)
{
        objLoc.UseProfileOf = objRole;
}
else
{
        TRIMSDK.SecurityLevel objSec = db.GetSecurityLevel("Confidential");
        objLoc.SecLevel = objSec;
```

```
        objLoc.UserType = utUserTypes.utRecordsWorker;
}
// Email address
TRIMSDK.LocEAddress objEmail = objLoc.LocEAddresses.New();
objEmail.EAddressType = etEAddressType.etMail;
objEmail.EAddress = "david@gbrmpa.com.au";
objEmail.Description = "Default business email";
// Relationships
objLoc.AddRelationship(objRole, lrLocRelationshipType.lrHasGroups,false);
TRIMSDK.Location objUnit = db.GetLocation("Administration");
objLoc.AddRelationship(objUnit, lrLocRelationshipType.lrMemberOf, true);
TRIMSDK.Location objBoss = db.GetLocation("Neumann, Ilse*");
objLoc.AddRelationship(objBoss, lrLocRelationshipType.lrBossedBy,false);
// Confirm & Save
if (objLoc.Verify(true))
{
        objLoc.Save();
        MessageBox.Show( objLoc.FormattedName + " created.");
}
else
{
        MessageBox.Show( objLoc.ErrorMessage);
}
```

Reference

Objects

The reference section detailing the methods and properties of each Content Manager COM SDK object has been replaced by helpstrings which appear in the object browser of your chosen IDE. These helpstrings contain the most up-to-date information about each method and property in the Content Manager COM SDK.