TRIM Context Software Development Kit (SDK)

March 2004
Using the TRIM Context Software Development Kit

Introduction

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

For those wanting to understand the capabilities of the SDK, this document can be read on its own. For those intending to use the SDK it serves as an orientation and introduction. For a complete technical understanding it should be used in conjunction with the Reference documentation, which provides detailed information on each object, method and property in the COM Interfaces and the ActiveX controls of the SDK.

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

Technical Prerequisites & Assumptions

The TRIM Context SDK components are based on Microsoft's Component Object Model (COM) standard. The SDK 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 with the SDK. The Visual Basic examples are tested using the Visual Basic 6 compiler, while the C# examples are tested using Microsoft Visual Studio .NET 2003, requiring the .NET Framework Version 1.1.

Using TRIM Context SDK with .NET applications

The TRIM Context Primary Interop Assemblies are integral to the function of the TRIM Context SDK with .NET applications. These are distributed as .DLL files and contain the .NET equivalent of the Type Libraries of the TRIM Context SDK.

When developing a new application in .NET, the Assemblies may be referenced from Visual Studio .NET through right-clicking on References in the Solution Explorer and browsing to select the necessary .DLL files.

A Short History of the SDK

In the early days of TOWER Software's TRIM Records & Document Management application (up to version 4.1), 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 TOWER released version 4.2 of TRIM 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, known as "Context", 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.

The new SDK COM interfaces (of which there are three) are in-process servers implemented in DLLs.

**What is the TRIM Context SDK?**

The TRIM Context SDK is a suite of tools that allow programmers to create custom solutions, services and integrated applications by leveraging the functionality of TRIM. These tools give TRIM 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.

At the core of the TRIM Context SDK is a comprehensive set of COM interfaces allowing programmatic access to the objects, data and business rules of the application.

In addition to the SDK exposing TRIM's data and functionality, Context includes some other great features for the integration developer. These include ActiveX controls for bringing the rich experience of the TRIM User Interface into custom applications, and an event processor interface for responding to key events as they occur in TRIM.

**Better Building Blocks**

TRIM Context has been designed in such a way that all the business objects used in the application are automatically exposed through a thin COM wrapper created automatically during the application build process. This means that using the TRIM Context SDK is very efficient, and all the same underlying objects and properties that the application’s programmers use to build the TRIM user interface are also available to third-party programmers via the SDK.

**Hear, Say**

As an Automation *server*, the TRIM 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 SDK is the set of questions that TRIM 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 Context SDK includes a feature that emancipates TRIM from this abject slavery, and gives it the opportunity to speak for itself. The `TRIMEventProcessor` is a special class the TRIM Object Model. It is an outbound interface, meaning that TRIM calls methods on this interface in response to events that occur in TRIM. 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 TRIM client.

**Taking Controls**

Another feature of the SDK that extends TRIM's integration options is the set of ActiveX controls. These are core elements of the TRIM application's user interface that can be built directly into another application's interface. The SDK includes methods that allow the programmer to invoke certain TRIM dialogs with the appearance of being 'hosted' by another application, but the ActiveX controls take the concept further, allowing edit controls, record lists and document viewers to be directly embedded inside custom dialogs and forms at design time.

**Software Development Kit (SDK)**

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

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

The SDK is an in-process server implemented in a dynamic link library (dll) file. This method of implementation means that the 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.

Future releases of TRIM will add two more SDKs – one for the TopDrawer utility and one for the new Workflow module.

**Event Interface**

An interface to notify important events in the TRIM SDK gives opportunities for integration programmers to control the level of synchronisation between TRIM and other enterprise applications. The SDK objects in general allow custom code to manipulate TRIM data and functionality on demand. The `TRIMEventProcessor` interface, on the other hand, allows custom code to run in response to common events that are invoked by TRIM.

Although the `TRIMEventProcessor` class is defined in the TRIMSDK Type Library (along with all the other TRIM 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 TRIM. (All other classes in the Type Library represent interfaces implemented by TRIM, and the programmer can call the methods on these interfaces).
The range of events that the event interface will notify 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.

**ActiveX Controls**

In addition to the SDK’s, which allow direct access to TRIM’s data and functionality, the Context SDK makes available certain components of the user interface as ActiveX controls. These controls are core elements of TRIM’s user interface that can be built directly into another application’s interface at design time.

The ActiveX controls have methods and properties that the programmer can access to control the behavior and appearance of each instance of a control at run-time. They also fire events to notify their 'host' of user actions such as data entry and mouse clicks on the control, so that the application can take appropriate action.

There are three TRIM controls included in the SDK:

- **Document Viewer**
- **Edit Box**
- **Tree Box**

The controls are contained in a component called "TRIM Context ActiveX Controls" and implemented by the file "tsjOCX.dll"; the methods and properties are defined in the Type Library "TRIMOCXLib".

**Document Viewer - TRIMviewer**

The Viewer control enables an application to view a wide variety of file formats without requiring the file's native application. The viewer control can be sized and positioned according to the design requirements of the host application. Any file on the file system can be viewed by the control (not just TRIM documents), so this can be very useful as a preview control.

**Edit Box - TRIMedit**

An Edit Box is a standard control in Windows applications that is used to allow a user to type in or edit data. TRIM makes extensive use of edit boxes to capture user input for record metadata and other information. In many cases, TRIM enhances the standard edit box by providing a 'KwikSelect' button integrated into the control, to allow the user to interactively select an object from the TRIM database, or a date from a calendar, or a file or directory from the local file system, instead of having to type the correct value manually. Many TRIM edit boxes also provide a history list, where the user's most recent selections of the same category (e.g. container files) are available in a drop-down list attached to the control.

TRIM provides its own enhanced edit box control in the Context SDK, so programmers and application designers can take advantage of the features it provides in custom applications and dialogs. The control can be set to be one of the following variations:

- **Custom Browser**
- **Date Picker**
- **Date and Time Picker**
- **Input File Selector**
• Output File Selector
• Directory Selector
• Spelling Checker
• Format Checker

The default behavior is pre-programmed into the control in all modes except Custom Browser and Format checker modes. That is, if the control is set at design-time to be a Date Picker, at run-time the control will display a calendar to the user when the KwikSelect is pressed, and the selected date will be entered into the text area of the control. No code is required by the client programmer to provide this functionality. Similarly, if the control is set as an Input File selector, the standard File Open dialog is displayed for the user to select a file from their file system, and the name of the selected file is automatically entered in the text of the control.

If the control is set to be a Custom Browser or Format Checker then the programmer will need to write code in response to the Browse event, which will be fired by the control when the user presses the KwikSelect button. This is because the required behavior will depend upon the context in which the control is being used.

Tree Box - TRIMtreeBox

The TRIM Tree Box control is used to display tabular data, i.e. rows of records with columns of record attributes. The rows may also be related in a hierarchy or tree structure, in which case the control will show the data rows in a Windows Explorer-style tree that can be navigated by the user expanding and collapsing branches.

The tree control uses enhanced column headers, which can be added, removed and reordered programmatically or interactively by the user. The headers can be resized manually or automatically to fit the data contents, and the user can sort the data by any column in ascending or descending order simply by clicking on a heading.

Like the other Context ActiveX controls, the Tree control does not require a connection to a TRIM database, and it can be used to display non-TRIM data. The control is configured and populated by passing text strings to methods of the control. Many of TRIM's SDK objects have special methods that are designed specifically for the Tree control, converting their data into appropriately formatted strings.

The TRIM Object Model

Objects and Interfaces

To understand how objects in the TRIM Context 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 TRIM 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 TRIM 4.3, all objects had a single interface, and all methods and properties that the object implemented were accessed through that interface. In the TRIM Context SDK, things are slightly different, as 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 SDK, such that new types of objects can be introduced in later versions, without breaking the existing interface.

The common interfaces in the SDK are:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implemented By</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBaseObject</td>
<td>Any objects that can be created and deleted independently.</td>
</tr>
<tr>
<td>IBaseChildObject</td>
<td>Any object that can only exist as a 'child' of a Base object.</td>
</tr>
<tr>
<td>IBaseObjects</td>
<td>Any collection of Base objects.</td>
</tr>
<tr>
<td>IBaseChildObjects</td>
<td>Any collection of Base Child objects.</td>
</tr>
</tbody>
</table>

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 TRIM 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 TRIM 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 TRIM 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 in the Context object model 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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Returns the Database object that created this object.</td>
</tr>
<tr>
<td>Uri</td>
<td>Returns the internal number that uniquely identifies this object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Verified</td>
<td>Returns True if the current state of the object's data is valid.</td>
</tr>
<tr>
<td>Type</td>
<td>Returns the object type for this object (e.g. Record, Location etc.)</td>
</tr>
<tr>
<td>ErrorMessage</td>
<td>Returns the description of the last error associated with this object.</td>
</tr>
</tbody>
</table>

**Common Methods**

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetProperty</td>
<td>Returns the data value of a property identified by a property Id.</td>
</tr>
<tr>
<td>SetProperty</td>
<td>Sets the data value of a property identified by a property Id.</td>
</tr>
<tr>
<td>GetPropertiesAsString</td>
<td>Returns the data values of a set of properties for the object, as a string formatted for the use specified.</td>
</tr>
<tr>
<td>Verify</td>
<td>Checks the validity of the current state of the object's data.</td>
</tr>
<tr>
<td>Save</td>
<td>Saves the current state of the object to the TRIM database</td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes the object from the TRIM database</td>
</tr>
</tbody>
</table>

**Interactive Methods**

Most methods in the Context object model allow the programmer to automatically perform some sort of data transformation in TRIM 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 TRIM objects, and therefore requires a TRIM dialog to be displayed. The Context object model exposes various methods that allow the programmer to invoke standard TRIM 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 zero instead, in which case TRIM will place the dialog in front of the current foreground window of the current active application.

Note that an alternative to calling interactive methods to achieve a TRIM 'look-and-feel' in an integrated application is to use the Context ActiveX controls within the user interface of the application. This is described later in the Context ActiveX Controls section.
See Next Page
Navigating the Object Model

Database Object

The Database object is the top-level object in the TRIM object model hierarchy. Because all objects in TRIM can only exist in the context of a database, the Database object is responsible for accessing and creating all other persistent business objects in the TRIM SDK. (There are certain helper objects in the object model that do not need a database, such as InputDocument, SignatureTool and EnumHelper. These non-database specific objects are discussed later in this document).

The Database object is listed at the top of the object list in the reference section of this help file. All other objects are listed in alphabetical order.

Connecting

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 TRIM security profile when accessing any TRIM 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:

```
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, TRIM will automatically attempt a connection.

In Visual Basic:

```
Set objTRIM = New TRIMSDK.Database
' Connection is automatic when required if not explicit
Set objRec = objTRIM.GetRecord(123)
```

In C#:

```
TRIMSDK.Database db = new TRIMSDK.Database();
// Connection is automatic when required if not explicit
TRIMSDK.Record objRec = db.GetRecord(123);
```

If a different (non-default) database is required, the Databases collection can be used to select a specific database.

In Visual Basic:

```
Dim objTRIM as TRIMSDK.Database
Dim coDBs as New TRIMSDK.Databases
If coDBs.Count > 1 Then
    ' Let user select a database in a dialog
    Set objTRIM = coDBs.ChooseOneUI(hWnd)
End If
objTRIM.Connect
```

In C#:

```
TRIMSDK.Database db = null;
TRIMSDK.Databases coDBs = new TRIMSDK.Databases();
if (coDBs.Count > 1)
{
    int hWnd = Handle.ToInt32();
    // Let user select a database in a dialog
    db = coDBs.ChooseOneUI(hWnd);
}
if ( db != null )
```
Similarly, if a particular named database is required, this could also be selected programatically 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;
    }
}

**Accessing Persistent Objects**

There are generally two ways to access existing persistent objects in the TRIM 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.

**Visual Basic Example**

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

```
' 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
```
**C# Example**

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

```csharp
// Declare the object variable
TRIMSDK.Record objRecord;
string vntRecId;

// Determine the identifier
vntRecId = "RP95/1";

// Call Get... to instantiate the object
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.

**Visual Basic Example**

The following example code creates a new Keyword (thesaurus term) object.

```vbnet
' 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
```

**C# Example**

The following example code creates a new Keyword (thesaurus term) object.

```csharp
// Declare the object variable
TRIMSDK.Keyword objKeyword;

// Call New... to instantiate the object
objKeyword = db.NewKeyword();

// Set properties
objKeyword.Name = "Example";
objKeyword.TopTerm = true;

// Save to the Database
```
Working With Collections

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.

Visual Basic Example

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

```vb
Dim colRecTypes As RecordTypes
Dim objRecType As RecordType

Set colRecTypes = objTRIMdb.MakeRecordTypes
Call colRecTypes.SelectAll
Set objRecType = colRecTypes.ChooseOneUI(hWnd)
```

C# Example

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

```csharp
TRIMSDK.RecordTypes colRecTypes;
TRIMSDK.RecordType objRecType;

colRecTypes = db.MakeRecordTypes();
colRecTypes.SelectAll();
int hWnd = Handle.ToInt32();
objRecType = colRecTypes.ChooseOneUI(hWnd);
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectAll</td>
<td>Fill the collection with all the objects of its type from the database.</td>
</tr>
<tr>
<td>SelectByPrefix</td>
<td>Add items to the collection by name prefix.</td>
</tr>
<tr>
<td>SelectByUris</td>
<td>Add specific items to the collection. Takes an array of object URIs.</td>
</tr>
</tbody>
</table>
Working With Child Objects

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

The only base objects in the TRIM 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.

Visual Basic Example

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

```vbnet
Dim objRecord As Record
Dim colKeywords As RecKeywords
Set objRecord = objTRIMdb.GetRecord("RP95/1")
Set colKeywords = objRecord.RecKeywords
Call colKeywords.DisplayUI(hWnd)
```

C# Example

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

```csharp
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.

---

<table>
<thead>
<tr>
<th>Other &quot;Select...&quot; Methods</th>
<th>Add items by other criteria. Different collection types will implement different selectors.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RefineUI</td>
<td>Allow the user to select items using a search dialog. (Not all collections provide this method.)</td>
</tr>
</tbody>
</table>
3. Edit the properties of (and/or call methods on) the child object variable.
4. Save the parent object.

**Visual Basic Example**

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

```vbnet
Dim colContacts As RecLocations
Dim objContact As RecLocation

Set objRecord = objTRIMApp.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
```

**C# Example**

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

```csharp
TRIMSDK.RecLocations colContacts;
TRIMSDK.RecLocation objContact;
TRIMSDK.Record objRecord = db.GetRecord("G96/201");
colContacts = objRecord.RecLocations;
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.

<table>
<thead>
<tr>
<th>Record Method</th>
<th>Child Object Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttachContact</td>
<td>RecLocation</td>
</tr>
</tbody>
</table>
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.

Object Properties

This section describes how an object's data is manipulated via properties and methods.

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 (e.g. 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:

```vbnet
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 TRIM's internal property identifiers to specify a property (see the Property Ids section for a list of identifiers for the main objects), 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 TRIM 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 TRIM Context, the system administrator can define any number of additional user-defined fields to be associated with TRIM 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 section on the `FieldDefinition` object below.

**The PropertyDef object**

Generic handling of properties of all TRIM 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 `PropertyIds` for the main TRIM object classes), 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.

**Visual Basic Example**

This example 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.

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

C# Example

This example 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.

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

TRIM Context 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.

Common Scenarios

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

Accessing a Record

To read information stored on records in a TRIM 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 TRIM 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 TRIM. 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. For example, if you wish to instantiate the record 2002/0059, you need to use the following statement:

**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: TRIM stores the Record Number in two formats. The expanded format (e.g. "2002/0059") is held in the LongNumber property, and the compressed format (e.g. "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 TRIM. 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. See the following example:

**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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>&quot;G1997/0770&quot;</td>
</tr>
<tr>
<td>Title</td>
<td>&quot;Greenhouse Journal of Global Warming - Dugong Habitats&quot;</td>
</tr>
<tr>
<td>DateCreated</td>
<td>#8/20/1997#</td>
</tr>
<tr>
<td>ExternalId</td>
<td>&quot;GJGW 97PB&quot;</td>
</tr>
<tr>
<td>AccessionNbr</td>
<td>5517</td>
</tr>
</tbody>
</table>

**Visual Basic Example**

```vba
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
```
C# Example
TRIMSDK.Database db = new TRIMSDK.Database();
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 TRIM 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.

Visual Basic Example
The following code instantiates a record object (in variable objRecord) and then assigns its container to another variable (objContainer).
Dim objRecord As Record
Dim objContainer As Record
Set objRecord = objTRIM.GetRecord ("G99/15")
Set objContainer = objRecord.Container ' objContainer is now 97/1004

C# Example
The following code instantiates a record object (in variable objRecord) and then assigns its container to another variable (objContainer).
TRIMSDK.Record objRecord = db.GetRecord ("G99/15");
// objContainer is now 97/1004

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

- CurrentLoc – current location of the record
- HomeLoc – normal location of the record
- OwnerLoc – location of the owner or responsible unit for the record
- AuthorLoc – person who authored the electronic document
- CreatorLoc – person who registered the record in TRIM
- AddresseeLoc – person to whom the record is addressed
- PrimaryContactLoc – the main contact person (or organization) for the record.

To access the properties and methods of these location objects, you can create and instantiate them using the following style of code:

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 TRIM. The SDK also allows you to update TRIM 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 the section on 'Verifying' below.) For more complicated types of update to a record, you must generally call methods that instruct TRIM to modify the record, based on arguments passed.

Modifying Properties

The simplest way to update data in a TRIM 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, AddressLoc and OtherLoc) and miscellaneous properties such as External Id, Priority, Accession Number and Foreign Barcode.

Visual Basic Example

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

C# Example

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 and Error Trapping

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 that the Date Registered is not in the future. If a property update cannot be carried out because of field-level verification, the method call or property assignment 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 is performed when the object’s Verify or Save methods are called.

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 below) 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.

Visual Basic Example
  If Not objRecord.Verify(True) Then
    MsgBox objRecord.ErrorMessage,,"Verify Failed"
  Else
    objRecord.Save
  End If

C# Example
  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 (see below) 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 TRIM 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.
Visual Basic Example

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

' 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

C# Example

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

// 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 yet been checked for internal consistency. Verified property is set to: ",
        & objRecord.Verified);
    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);
{ objRecord.Title = "new Title";
   MessageBox.Show("The title of the Record has just been changed. The Record has
t 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 on 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.

Visual Basic Example

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

C# Example

TRIMSDK.Record objRecord = db.GetRecord("G97/770");
objRecord.Title = objRecord.Title + " plus New Part of Title";
DateTime dateDue = new DateTime(2003,1,1);
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.

Visual Basic Example

```vbnet
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
```

C# Example

```csharp
// 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.)

Visual Basic Example

```vbnet
Dim objSearch As RecordSearch ' declare the search object
Set objSearch = objTRIM.NewRecordSearch ' make the object
Call 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.

For example, to retrieve records that contain the word "reef" within the title, you would add a Title Word clause passing the argument "reef", as follows:

**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#:**
```
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)'.

Another example, this time using 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();
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.

Visual Basic Example
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

C# Example
TRIMSDK.RecordSearch objSearch = db.NewRecordSearch();
objSearch.AddTitleWordClause("Press");
objSearch.AddDateRegisteredClause((DateTime.Today.AddDays(-1), DateTime.Today);
objSearch.AddLocationClause(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...'

Visual Basic Example
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

C# Example
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 example sorts the results by ascending Priority, then Record Type, then descending Date Due.

**In Visual Basic:**

\[
\text{Call objSearch.Sort(rsPriority,,rsRecordType,,rsDateDue, True)}
\]

**In C#:**

```
objSearch.Sort(rsRecordSortFields.rsPriority,false,rsRecordSortFields.rsRecordType,false,rsRecordSortFields.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 next section) 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.

**Visual Basic Example**

```
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
```

**C# Example**

```
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, to select multiple records or simply to browse the results for viewing. See the documentation for the Records collection object for details.

**Processing Results Sequentially**

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

**Visual Basic Example**

```
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.GetNext
Do Until objRecord Is Nothing
    iMonth = Month(objRecord.DateCreated)
    sCosts(iMonth) = sCosts(iMonth) + objRecord.GetUserField(objCost)
    Set objRecord = objSearch.GetNext
Loop

C# Example

In this example, all records returned by the search are processed by adding up
the values in a user-defined field called 'Actual Cost', subtotalled by month based
on the date the record was created.

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.GetNext();
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();
}

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 TRIM Context 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, 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:**

```vbnet
' Create a new Case File record
Set objRecType = objTRIM.GetRecordType("Case File")
Set objRecord = objTRIM.NewRecord(objRecType)
```

**In C#:**

```csharp
//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 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 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.

**Visual Basic Example**

```vbnet
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"
```

**C# Example**

```csharp
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".

**Visual Basic Example**
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 TRIM Help – Administrator Guide – 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.

Visual Basic Example

```vbnet
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"))
```

C# Example

```csharp
// 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 code produces the same result as the example above:

Visual Basic Example

```
' Assign security level and two caveats
objRecord.Security = "Confidential, Research Projects, Staff in Confidence"
```

C# Example

```
// 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.
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.)

**Visual Basic Example:**

This example grants the following:

- 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.

```vbnet
Call objRecord.SetAccessControlDetails(dxViewRecord, asPublic)
Call objRecord.SetAccessControlDetails(dxUpdateMetadata, asInherited)
Call objRecord.SetAccessControlDetails(dxDeleteRecord, asPrivate, objTRIM.GetLocation("Records Manager"))
```

**C# Example:**

This example grants the following:

- 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.

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

**Visual Basic Example**

This line of code creates a relationship of "Record A supersedes Record B".

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

**C# Example**

This line of code creates a relationship of "Record A supersedes Record B".

```csharp
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 below).

*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.)

**Visual Basic Example**

This example sets the record's Home location to the unit called "Administration", and the Current location to the connected user.

```vbnet
objRecord.SetHomeLocation(objTRIM.GetLocation("Administration"))
objRecord.SetCurrentLocation(objTRIM.CurrentUser)
```

**C# Example**
This example sets the record's Home location to the unit called "Administration", and the Current location to the connected user.

```vbnet
objRecord.SetHomeLocation(db.GetLocation("Administration"));
objRecord.SetCurrentLocation(db.CurrentUser, DateTime.Now);
```

### Record Contacts

Unlike record locations (see above), 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'.

#### Visual Basic Example

This example sets the record's Representative (and primary contact) to be the connected user, and the Client to be the organisation called "Tower".

```vbnet
objRecord.AttachContact(objTRIM.CurrentUser, ctRepresentative, True)
objRecord.AttachContact(objTRIM.GetLocation("Tower"), ctClient)
' ctClient = Client
```

#### C# Example

This example sets the record's Representative (and primary contact) to be the connected user, and the Client to be the organisation called "Tower".

```csharp
objRecord.AttachContact(db.CurrentUser, ctContactType.ctRepresentative, true, DateTime.Now);
objRecord.AttachContact(db.GetLocation("Tower"), ctContactType.ctClient, false, DateTime.Now); // ctClient = Client
```

#### Code Example – Visual Basic

This code demonstrates many of the features described above. The code will work with the Demonstration database provided on the TRIM installation disk.

```vbnet
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) ' ctClient = Client
' Verify and Save
If Not .Verify Then
MsgBox .ErrorMessage
Else
.Save
End If
```
End With

**Code Example – C#**

This code demonstrates many of the features described above. The code will work with the Demonstration database provided on the TRIM installation disk.

```csharp
TRIMSDK.Database db = new 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 // must enter Description
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); // ctClient = Client

// 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 the previous section for details of [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 electronic 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.)

**Visual Basic Example**

```vbnet
objRecord.Title = "Letter from executor regarding disbursements of Lady Marchcroft's bequest"
objRecord.LongNumber = "XK/008934"
```

**C# Example**

```csharp
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.

**Visual Basic Example**

```vbnet
Dim objContainer As Record
Set objContainer = objTRIM.GetRecord("76/915")
objRecord.SetContainer(objContainer, True)
```

**C# Example**

```csharp
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.

**Visual Basic Example**

```vbnet
Dim objDoc As New InputDocument
Call objDoc.SetAsFile("C:\myDocs\ThisFile.doc")
Call objRecord.SetDocument(objDoc, False, False, "Created via SDK")
```

**C# Example**

```csharp
TRIMSDK.InputDocument objDoc = new InputDocument();
// note that in C# the \ character is an escape symbol,
```
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.

**Visual Basic Example**

```vbnet
If Not objRecord.SetDocumentUI(hWnd, "TheDefault.doc", "Attach Document", False) Then
  MsgBox "Action cancelled."
Exit Sub
End If
```

**C# Example**

```csharp
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.

**Visual Basic Example**

This example sets the document's Author to be the connected user.

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

**C# Example**

This example sets the document's Author to be the connected user.

```csharp
objRecord.AuthorLoc = db.CurrentUser;
```

**Access Control for Documents**

For more information on this subject, see Access Control earlier in this document.

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.

**Visual Basic Example**

This example grants the following:

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

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

**C# Example**

This example grants the following:

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

```csharp
objRecord.SetAccessControlDetails(dxRecordAccess.dxUpdateDocument, asAccessControlSettings.asPrivate, db.CurrentUser);
```
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 *The FieldDefinition Object* earlier in this document.)

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.

**Visual Basic Example**

This example 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.

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

**C# Example**

This example 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.

```csharp
objRecord.SetUserField(db.GetFieldDefinition("Job Code"), "D0933");
```

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.

**Visual Basic Example**

This example 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.

```visualbasic
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
```

**C# Example**

This example 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.

```csharp
TRIMSDK.Record objContainer = db.GetRecord("96/715");
TRIMSDK.Records colContents = db.MakeRecords();
colContents.SelectContentsOf(objContainer);
int hWnd = Handle.ToInt32();
```
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.

**Visual Basic Example**

```vbnet
If objDoc.IsElectronic Then
    Call objDoc.GetDocument("C:\tmp\MyFile.doc", True, "Checked out via SDK")
End If
```

**C# Example**

```csharp
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.

**Visual Basic Example**

In this example, the user selects from their list of favorite documents.

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

**C# Example**

In this example, the user selects from their list of favorite documents.

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

Check In

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.

**Visual Basic Example**

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

**C# Example**

```csharp
{
    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.

**Visual Basic Example**

```vbnet
If Not SetDocumentUI(hWnd, "MyFile.doc", "Check-In", False) Then
    MsgBox "Check-in cancelled"
End If
```
C# Example
```csharp
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.

**Visual Basic Example**
```vb
Dim objLoc As Location
Set objLoc = objTRIM.GetLocation("Abbott, Peter (Mr)")
```

**C# Example**
```csharp
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.

**Visual Basic Example**
```vb
Set objLoc = objTRIM.GetLocation("Abbott, P"")
Set objLoc = objTRIM.GetLocation("Abbott"")
Set objLoc = objTRIM.GetLocation("Abbott, Peter""")
```

**C# Example**
```csharp
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.

**Visual Basic Example**
```vb
Set objLoc = objTRIM.GetLocation("Abb"""); ' finds Abbott and Abbey
If objLoc Is Nothing Then Exit Sub
```

**C# Example**
```csharp
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.

Visual Basic Example

```visualbasic
Dim objUnit As Location
Dim objjBoss 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

C# Example
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@gbbrmpa.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);
    }
What's New from TRIM Captura to the TRIM Context SDK

This section describes the main differences between the TRIM Context (5.0) interfaces and those of the TRIM Captura (4.3), for the benefit of programmers required to upgrade code developed against one to the other.

**Summary of Changes**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Captura</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>Tsapi.exe Out of process</td>
<td>Trimsdk.dll In process</td>
</tr>
<tr>
<td>Type Library name &amp; description</td>
<td>tsapi TRIM 4.3 API Type Library</td>
<td>TRIMSDK TRIM SDK Type Library</td>
</tr>
<tr>
<td><strong>Top level object</strong></td>
<td>Application object</td>
<td>Database object</td>
</tr>
<tr>
<td><strong>Object names</strong></td>
<td>ITS_ prefix (e.g. ITS_record)</td>
<td>No prefix (e.g. Record)</td>
</tr>
<tr>
<td><strong>Database connection</strong></td>
<td>Explicit app.Connect(usr, pwd, db)</td>
<td>Automatic to default DB db.Connect</td>
</tr>
<tr>
<td><strong>Instantiating objects</strong></td>
<td>obj = Make&lt;object&gt;</td>
<td>obj = Get&lt;object&gt;</td>
</tr>
<tr>
<td><strong>Instantiating objects by URI</strong></td>
<td>obj.Lookup(123)</td>
<td>obj = GetRecord(123)</td>
</tr>
<tr>
<td><strong>Instantiating objects by name</strong></td>
<td>obj.Open(&quot;97/1004&quot;)</td>
<td>obj = GetRecord(&quot;97/1004&quot;)</td>
</tr>
<tr>
<td><strong>Instantiating Collections</strong></td>
<td>col = Make&lt;objects&gt;</td>
<td>col = Make&lt;objects&gt;</td>
</tr>
<tr>
<td><strong>Create new base object</strong></td>
<td>obj = New&lt;object&gt;</td>
<td>obj = New&lt;object&gt;</td>
</tr>
<tr>
<td><strong>Edit object in dialog</strong></td>
<td>obj/Edit</td>
<td>obj.PropertiesUI</td>
</tr>
<tr>
<td><strong>Verify Save OK</strong></td>
<td>If Not obj.Save Then MsgBox obj.ErrorMessage</td>
<td>obj.Verify If obj.Verified Then obj.Save</td>
</tr>
<tr>
<td><strong>Record Type</strong></td>
<td>ITS_recordType</td>
<td>RecordType</td>
</tr>
<tr>
<td>Select Record Type</td>
<td>type.Select(hwnd, False)</td>
<td>types.SelectAll type = types.ChooseOneUI</td>
</tr>
<tr>
<td>Select electronic Record Type</td>
<td>type.Select(hwnd, True)</td>
<td>.SelectAll .SetFilter(tfElectronic) type = .ChooseOneUI</td>
</tr>
<tr>
<td>Determine view</td>
<td>Use ITS_recordType object</td>
<td>Use PropertyDefs object</td>
</tr>
<tr>
<td>pane attributes</td>
<td>GetNextViewPaneAttribute</td>
<td>SelectViewPanelItems</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Record</td>
<td>ITS_record</td>
<td>Record</td>
</tr>
</tbody>
</table>

**Create new Record**

- `rec = NewRecord(recType)`
- `rec.SetTitle("A Title")`
- `rec.SetNumber("2002/054")`
- `rec.Create`

**Change Type**

- [Not Possible]

**Check-in Electronic Document**

- `rec.SetDocument(strFile)`
- `rec.SetDocumentUI`, `rec.SetDocument`

**Attributes by name**

- `rec.GetAttribute`
- `rec.SetAttribute`

**User defined fields**

- 15 properties, e.g. `rec.Field1`, `rec.Date3`
- Multiple definable properties. `rec.GetUserField`
- `rec.SetUserField`

**Catalogue Mail Messages**

- Use `ITS_mailMessage object`
- `rec.CreateFromMailMessage`

**Relate Records**

- `rec.SetRelated`
- `rec.AttachRelationship`

**Browse Related Records**

- Use `ITS_search object`
- `srch.RelatedTo(rec.Uri)`

**Location**

- ITS_location
- Location

**Location types**

- lcName, lcContact
- lcUnit, lcOrganization
- lcPosition
- lcPerson
- lcOrganization
- lcPosition
- lcGroup
- lcUnknown

**Get Current User**

- loc.Open("%ME%")
- `loc = db.CurrentUser`

**Create new external contact locations**

- NewContact
- NewLocation
- LocType = lcPerson
- IsWithin = False

**Create new staff within a unit**

- NewStaff(objUnit)
- NewLocation
- LocType = lcPerson
- AddRelationship(objUnit, lrMemberOf)

**Search**

- ITS_search
- RecordSearch

**Add Search clause**

- `srch.<clause>`
- eg: `srch.TitleWord`
- `srch.Add<clause>Clause`
- eg: `srch.AddTitleWordClause`

**Sorting**

- Fields specified as strings
- Fields specified as enum values

**Search Criteria dialog**

- `srch.Edit`
- `srch.EditQueryUI`

**Copy search results to record**

- `srch.Fill(recs)`
- `recs = srch.GetRecords`
**Summary of New features**

**New Objects**
- Action Definition
- Thesaurus Keywords
- Classification (Record Plan)
- Lookup Set & Lookup Item
- Electronic Store
- Field Definitions
- Property Definitions
- Census
- Litigation Hold
- History
- HTML Publishing Layouts
- Address
- Record Locations
- Record Relationships
- Renditions
- Revisions
- Requests
- Reports
- Schedules & Triggers
- Space Management
- Digital Signatures
- Security Levels & Caveats
- Zip Codes (Postal Codes)
- Workflow
New Concepts
Verification
Selecting for Collections
Making Reference Files
Property Definitions
Data Strings
Publishing to HTML
Child Lists
Enumeration Helper
Renaming Captions
System/Database Options

New Features In Detail

Objects
There are vastly more objects in the Context SDK. TRIM Captura's API object model consists of 7 objects, 4 collections and 9 enumeration groups. Context, on the other hand, has 43 objects, 39 collections and 62 enumeration groups.

When converting Captura API code to Context, all TRIM object declarations will need to be updated, as the names of all objects represented in the API have changed in Context.

The Type Library name in Captura was "tsapi"; in Context it is "TRIMSDK". You will only need to use this name as an object qualifier if you have other object libraries referenced in your project with similar object names.

‘4.3 Captura
Public objTRIM As New tsapi.Application
Public objRecord As ITS_record

‘5.0 Context
Public objTRIM As New TRIMSDK.Database
Public objRecord As Record

The equivalent objects and their correct names are tabulated below.

<table>
<thead>
<tr>
<th>Captura (tsapi)</th>
<th>CapturaContext (TRIMSDK)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Database</td>
</tr>
<tr>
<td>ITS_recordType</td>
<td>RecordType</td>
</tr>
<tr>
<td>ITS_recordTypes</td>
<td>RecordTypes</td>
</tr>
<tr>
<td>ITS_record</td>
<td>Record</td>
</tr>
<tr>
<td>ITS_records</td>
<td>Records</td>
</tr>
<tr>
<td>ITS_search</td>
<td>RecordSearch</td>
</tr>
</tbody>
</table>
For new objects, see the Context object model.

**Connecting to TRIM**

The TRIM 4.3 API required explicit declaration of a user name, password and database when connecting to TRIM. The programmer's options were to call the `Connect` method to specify these arguments in code, or to call `ConnectTD` (or `GetLoginDetails` then `Connect`) to get these details from a user in a dialog.

Context manages the connection to the database through the users' operating system login and the default database. The Database object's `Connect` method will attempt to connect the current user to their default database. It does not require any parameters.

```vbnet
Set objTRIM = New TRIMSDK.Database
objTRIM.Connect
```

For more details see Connecting.

**Instantiating Objects**

Instantiating existing objects in Captura involved calling 'Make<object>' methods on the Application object to return an object, then passing the URI or Name of the required object to a Lookup or Open method.

In Context, you can instantiate the object at the same time as you create the object, by passing a name or URI value to a 'Get<object>' method on the Database object. The 'Get...' methods accept a variant parameter for the name or URI of the object.

### 4.3 - Change title of Record number 97/1004

```vbnet
Dim objRec As ITS_record
Set objRec = objTRIM.MakeRecord
If Not objRec.Open("97/1004") Then
    MsgBox objRec.ErrorMessage
    Exit Sub
End If
objRec.SetTitle("New Title")
objRec.Save
```

### 5.0 - Change title of Record number 97/1004

```vbnet
Dim objRec As Record
Set objRec = objTRIM.GetRecord("97/1004")
If objRec Is Nothing Then
    MsgBox "Record not found"
    Exit Sub
End If
objRec.Title = "New Title"
objRec.Save
```
Creating an Electronic Record

'4.3 'Select the Record Type
Set DocType = app43.MakeRecordType
If Not DocType.Select(hwnd, True) Then
    MsgBox DocType.ErrorMessage, , "TRIM Image Scanner"
    Set DocType = Nothing
    Exit Sub
End If

'Initialise the new Record
Set detail = app43.NewRecord(DocType) ' Create a new record

'Save/Create the New Record
If detail.Create(hwnd, TopForm.ctlImgEdit.Image, True) Then
    MsgBox "Created. Record Id: " + detail.Number
End If

The TRIM 5.0 SDK introduces a new object called an Input Document. It allows the electronic details of a record to be set up before creating the record. It has two forms, a file object or a mail message object. Pass the InputDocument object as an argument to the SetDocument method. The properties of the new record can be viewed and edited using the PropertiesUI method of the Record. Call Save to persist the Record to the Database.

'5.0 'Select the Record Type
Dim DocTypesCollection As TRIMSDK.RecordTypes
Dim DocType As TRIMSDK.RecordTypes
Set DocTypesCollection = app50.NewRecordTypes

' Fill the Collection with all Record Types
DocTypesCollection.SelectAll
' Filter the Collection, only include those RecordTypes that support the creation of electronic records
DocTypesCollection.SetFilter (rfElectronicForCreate)
Set DocType = DocTypesCollection.ChooseOneUI(hwnd)

'Initialise the new Record
Set detail = app50.NewRecord(DocType) ' Create a new record

Dim inDoc As New TRIMSDK.InputDocument
inDoc.SetAsFile (TopForm.ctlImgEdit.Image)

'Save/Create the New Record
If detail.SetDocument(inDoc) Then
    detail.PropertiesUI (hwnd)
    detail.Save
    MsgBox "Created. Record Id: " & detail.Number
End If

For more details on creating objects generally, see Creating a New Object.
What's New from TRIM Context to TRIM Context 5.2 SDK

This section describes the main differences between the TRIM Context (5.0) interfaces and those of TRIM Context (5.2), for the benefit of programmers familiar with TRIM Context (5.0) who want to know what's new.

New Objects

DocumentQueue – The ability to map a TRIM Folder to a directory or Lotus Notes folder
DocumentQueues
ExternalQueue
ExternalQueueItem
ExtractDocument – More control when extracting electronic documents from TRIM
ReportBitmap
ReportBitmaps
ScheduledTask
ScheduledTasks
SecurityGuide
SecurityGuides
TaskTemplate – Reserved for future use
TaskTemplates – Reserved for future use
TRIMFieldAddIn – Add-In type control on String type User defined fields assisting integrations
TRIMRecordAddIn - Add-In type control on the record object assisting integrations
**New Concepts**

**TRIM Add-Ins**

The [TRIMFieldAddIn](#) and **TRIMRecordAddIn** have been developed to enhance the ability to integrate with TRIM. The **TRIMEventProcessor** (available in earlier releases of TRIM Context) is used for any background processing you require when events are raised asynchronously by TRIM. For example the creation of TRIM objects creates events and when you catch the event (via the TRIMEventProcessor) you can do any processing you require. This works very well for any background tasks but there was a vision to provide the ability to give a greater level of interaction between some events and the user at the time the event occured. The results of this vision are the **TRIMFieldAddIn** and the **TRIMRecordAddIn** classes. You can now get control before a record properties dialog is displayed to the user and populate some default values based on your rules. Control is also available prior to a record being saved so you can do verification based on your rules. After a record is saved you can also take control to update any related information in TRIM or a different system based on your rules. Similar control is available for selecting possible values for a user defined field and verifying the value. All this happens client side with the ability to interact with the user when it is relevant to do so.

Extracting an electronic document from TRIM

In the past you have been able to get an electronic document out of TRIM via the `Record.GetDocument` method. This provided the ability to get the electronic document associated with a TRIM Record and optionally to mark it as checked out. The [ExtractDocument](#) provides more options and control of this process. It gives you the ability to investigate and select some options prior to doing the actual extract. For example you can get a 'browser friendly' version, check the size and type of the file and determine if the document contains OLE links.