Rocket U2 Clients and APIs

UniObjects for .NET Developer Guide

Version 5.3.0
(Component Version 3.171.1)

October 2022
UCC-530–UON-DG-01

Notices

Edition

Publication date: October 2022
Book number: UCC-530–UON-DG-01
Product version: Version 5.3.0(Component Version 3.171.1)

Copyright

Trademarks

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	400-120-9242
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Chapter 1: Introduction

This chapter introduces the UniObjects for .NET interface. It first gives you a basic understanding of Microsoft .NET and the .NET Framework. It then describes the architecture of UniObjects for .NET. Finally, it provides an overview of the features of UniObjects for .NET.

About UniObjects for .NET

UniObjects for .NET is an interface to the UniData and UniVerse databases through Microsoft .NET. UniObjects for .NET is a proprietary middleware application program interface (API) designed specifically for software development in the .NET Framework. This interface is managed code written in C# Common Language Runtime (CLR).

Software developers can use the UniObjects for .NET API and any CLR language (such as C#, J#, VB.NET, or C++ .NET) to create the following types of application and services:

- Console applications
- Windows applications
- ASP.NET Web applications
- XML Web services
- Smart Client applications for desktop and pocket PCs

You will need to know more about Microsoft .NET and the .NET Framework before we go on to discuss its use with UniObjects. The next section introduces you to the concepts and components of Microsoft .NET.

Note: The UniObjects for .NET driver does not support devices licensed on Windows Terminal service.

About Microsoft .NET

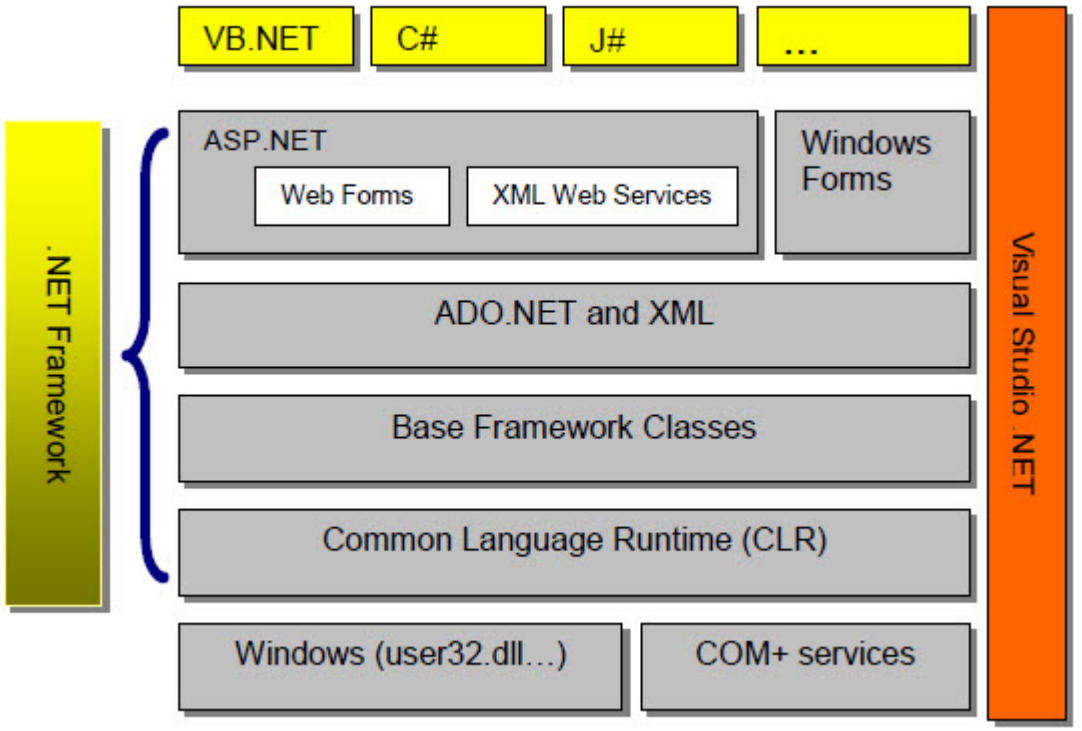
Microsoft .NET is a complete software package for developing and delivering software applications. It consists of the following components:

- .NET Framework, used to build and run several types of software, including Web-based applications, smart client applications, and Extensible Markup Language (XML) Web services. These types of software facilitate integration by sharing data and functionality over a network through standard, platform-independent protocols such as XML, SOAP, and HTTP.
- Developer tools, such as Microsoft Visual Studio® .NET 2013, which provide an integrated development environment (IDE) for maximizing developer productivity with the .NET Framework.
- Server software, including Microsoft Windows® Server, Microsoft SQL Server™, and Microsoft BizTalk® Server, that integrates, runs, operates, and manages Web services and Web-based applications.
- Client software, such as Windows or Microsoft Office, that helps developers deliver a user interface across a variety of devices.

What is the .NET framework?

The .NET Framework is an integral Windows component for building and running a new generation of software applications and Web services. It is composed of the Common Language Runtime (CLR) and a unified set of class libraries.

The following illustration shows the .NET Framework in the context of the Windows environment. The parts of the .NET Framework are shown within the bracket marked .NET Framework.



With the .NET Framework, it is easier than ever to build, deploy, and administer secure, robust, high-performing software applications. The .NET Framework:

- Supports more than 20 programming languages, including C#, J#, VB.NET, and C++ .NET.
- Manages much of the underlying code for software applications, enabling software developers to focus on the core business logic code.

Common Language Runtime (CLR)

The CLR is responsible for run-time services such as

- language integration
- security enforcement
- memory, process, and thread management

In addition, CLR has a role at development time when features such as lifecycle management, strong type naming, cross-language exception handling, and dynamic binding reduce the amount of code that the software developer must write to turn business logic into a reusable component.

Microsoft Intermediate Language (MSIL)

The .NET Framework compilers generate this CPU-independent instruction set for the use of the Common Language Runtime. Before MSIL can be executed, the CLR must convert it to native, CPU-specific code.

Managed code

This type of code is executed and managed by the .NET Framework's Common Language Runtime. Managed code must supply the instruction set necessary for the CLR to provide services such as memory management, cross-language integration, code access security, and automatic lifetime control of objects. All code that has been compiled in Microsoft Intermediate Language executes as managed code.

Unmanaged code

This type of code is executed by the operating system, outside the .NET Framework's Common Language Runtime. Unmanaged code must provide its own memory management, type checking, and security support, unlike managed code, which receives these services from the Common Language Runtime.

Class libraries

The .NET Framework uses a number of class libraries, listed below. Together, the class libraries provide a common, consistent development interface across all languages supported by the .NET Framework.

- **Base classes** provide standard functionality such as input/output, string manipulation, security management, network communications, thread management, text management, and user interface design features.
- **ADO.NET** classes enable developers to interact with data accessed in the form of XML through the OLE DB, ODBC, Oracle, and SQL Server interfaces.
- **XML classes** enable XML manipulation, searching, and translation.
- **ASP.NET** classes support the development of Web-based applications and Web services.
- **Windows Forms** classes support the development of desktop-based smart client applications.

IBMU2.UODOTNET is the namespace assigned to UniObjects for .NET Compact Framework for the UniData and UniVerse databases.

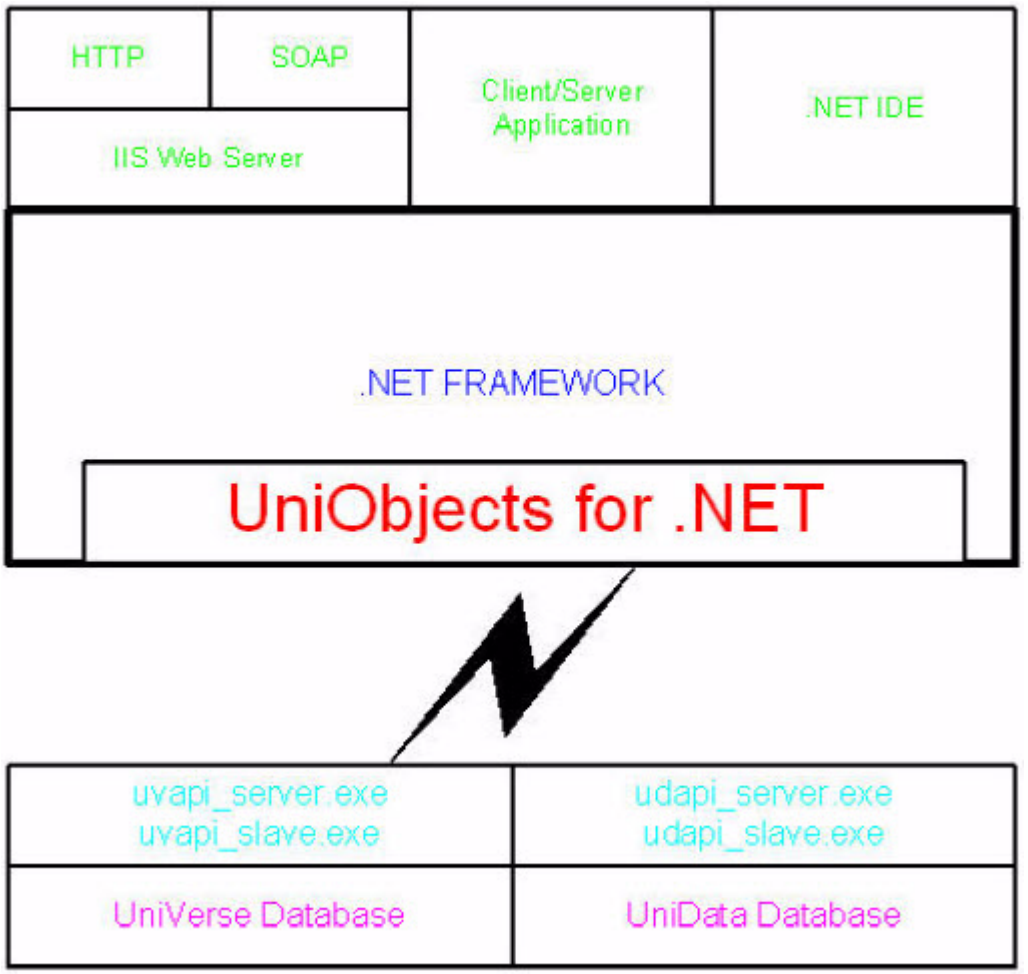
UniObjects for .NET Compact Framework is the data access model for .NET Compact Framework applications that connect to the UniData and UniVerse databases. It contains a collection of classes that allow you to connect to the UniData and UniVerse databases, execute commands, and read and write results:

- The UniSession class represents an open session to a UniData or UniVerse database.
- The UniFile and UniDictionary classes are used to access all file operations.
- The UniCommand class controls execution of database commands on the server. With it, users can run UniData or UniVerse commands or stored procedures on the server.
- The UniSubroutine class is used to execute cataloged BASIC server-side subroutine commands on the server.

- The UniTransaction class represents a Basic transaction to be made in a UniData or UniVerse database.
- The UniDataSet is a collection class used to read and write bulk UniRecord transactions in a UniData or UniVerse database.

Architecture of UniObjects for .NET

The following illustration shows the architecture of UniObjects for .NET and its relationship with the UniData and UniVerse databases.



U2.UODOTNET is the namespace assigned to UniObjects for .NET for the UniData and UniVerse databases.

UniObjects for .NET is the data access model for .NET applications that connect to the UniData and UniVerse databases. It contains a collection of classes that allow you to connect to the UniData and UniVerse databases, execute commands, and read and write results:

- The UniSession class represents an open session to a UniData or UniVerse database.
- The UniFile and UniDictionary classes are used to access all file operations.
- The UniCommand class represents a Basic statement or stored procedure to execute against a UniData or UniVerse database.

- The UniTransaction class represents a Basic transaction to be made in a UniData or UniVerse database.
- The UniDataSet is a collection class used to read and write bulk UniRecord transactions in a UniData or UniVerse database.

Features of UniObjects for .NET

UniObjects for .NET (UO.NET) is written in pure C#, one of the .NET Framework CLR-supported languages.

Since UniObjects for .NET is written purely in C#, it is managed code; it does not use any functions outside of the .NET Framework CLR. The UO.NET code complies with the .NET Framework standard and it follows C# conventions for names, comments, and standards.

UniObjects for .NET supports the IPv6 protocol, while maintaining IPv4 compatibility.

NLS support

UniObjects for .NET supports the Encoding class of the .NET Framework class library through its UniSession class property called UOEncoding.

The Encoding class provides methods to convert arrays and strings of UniCode characters to and from arrays of bytes encoded for a target page. An application can use the properties of the Encoding class such as ASCII, Default, Unicode, UTF7, and UTF8.

For example:

```
UniSession m_us=UniObjects("xxx", "yyy", "localhost", "demo", "udcs");
    Encoding en=Encoding.UTF8;
    m_us.UOEncoding=en;
```

Tracing and logging

UniObjects for .NET provides a standard tracing and logging facility to trace the execution of UO.NET code and log the data in a user-specified destination. The configuration of UO.NET tracing and logging can be specified in your application's configuration file (app.config for Windows applications or web.config for Web applications or Web services).

In the .NET Framework, there are four predefined trace levels:

- 1(error)
- 2(warning)
- 3(info)
- 4(verbose)

A UniObjects for .NET application can select one of these four levels and specify a storage destination for the output of tracing and logging. If no destination is given, the default output file UniTraceLog.txt is generated in the system's temporary folder.

By default, tracing and logging are turned off in UniObjects for .NET. Tracing can be turned on using the application configuration file. For example:

```
<system.diagnostics>
```

```
    <switches>

        <!-- Set value property of Arithmetic switch to one of the following:
            1(error), 2(warning), 3(info), 4(verbose) -->
        <add name='UniTraceSwitch" value="1" />
    </switches>
    <trace autoflush="true" indentsize="4">
        <listeners>
                        <add name="myListener"
                        type="System.Diagnostics.TextWriterTraceListener"
                        initializeData="c:\temp\myListener.log" />
        </listeners>
    </trace>

</system.diagnostics>
```

In the above example., tracing is turned on and it is set to the 1(error) level. The log file name is `c:\temp\myListener.log`.

As we expect UniObjects for .NET to be used in a multithreaded environment, thread ID and thread name are a standard prefix to tracking and logging messages.

UniDynArray and UniDataSet

In UniObjects for .NET, UniDynArray and UniDataSet are always associated with UniSession so they use the marks that the server is set up to use. Consequently, there is no chance of server data being misinterpreted due to the use of different marks on the client and server.

UniDynArray stores data internally in byte arrays, so it deals with binary data only, as does the server.

UniDataSet supports foreach statements. For example:

```
UniSession m_us=UniObjects("xxx", "yyy", "localhost", "demo", "udcs");
UniFile m_fl=m_us.Openfile("Customer");
string[] sArray={"2", "3", "4"};
UniDataSet m_ds=m_fl.Read(sArray);
foreach(UniRecord item in M_ds)
{
    Console.WriteLine(item.ToString[]);
}
```

UniFile read/write methods

To read a single field from a file, you can use `UniFile.ReadField()` or `UniFile.ReadNamedField()`; to read multiple fields, you can use `UniFile.ReadFields()` or `UniFile.ReadNamedFields()`.

The UniObjects for .NET UniFile Read and Write methods take parameters of specific types like int, string, and string[]. Strong typing methods have early binding requirements that can catch application errors during compilation rather than at run time.

Chapter 2: Using UniObjects for .NET

This chapter explains how to use UniVerse or UniData in a .NET application. The topics covered include:

- An overview of the database environment
- Opening and controlling a database session
- Accessing files
- Locking records
- Handling errors
- Using dictionaries
- Accessing UniVerse text files and binary files for sequential processing
- Executing database commands
- Running subroutines on the server
- Using encryption wallets

The database environment

This section tells you just enough about the database environment to enable you to understand the rest of the chapter.

If you already know about UniVerse or UniData, skip to UniObjects concepts. To learn more about UniVerse, read *UniVerse System Description*. To learn more about UniData, read *Using UniData* and *Administering UniData*.

A database user logs on to a database account. A database account includes an operating system directory containing database files and possibly operating system files and directories as well.

Note: UniVerse has several account flavors. The following sections describe the UniVerse IDEAL flavor, which is recommended for use with UniObjects. UniData uses ECLTYPE and BASICTYPE to specify account flavors. See the Using UniData manual for information about UniData flavors.

Each database file comprises a data file containing data records, and a file dictionary that defines the structure of the data records and how to display them. Each record in a file is uniquely identified by a record ID, which is stored separately from the data to which it refers.

The VOC (vocabulary) file in a database account contains a record for every file used in the database. This record provides a cross-reference between the file name, which is the record ID, and the path of the file stored in field 2 of the record.

Data structure

In an application, each file holds one type of record. For example, a file called CUSTOMER might hold one record for each customer, whereas another file called ORDERS might hold one record for each order placed by a customer. The records and the fields they contain are not fixed in size, and the file itself can grow or shrink according to the amount of data it holds.

Data is stored in fields in a record. For example, a record in the CUSTOMER file might have fields containing the name, address, and telephone number of a customer. A field can hold more than one value, for example, the separate elements of an address can be stored as multivalues of one field

rather than as separate fields in the record. A field in one record can contain a cross-reference to data held in another file. For example, to link customers with their orders, records in the CUSTOMER file might have a multivalued field containing a list of the corresponding record IDs of their orders in the ORDERS file.

File dictionaries

The file dictionary holds information about the structure of data records and their relationships to other files. In a record, each field is identified by a number, and the dictionary acts as a cross-reference between that number and the name of the field. For example, the customer's phone number might be held in a field called CUST.PHONE, which is field 3 in the record.

The file dictionary also defines how to format and display the data in the field for output; for example, the heading and the width of the column used in a report. All data is stored as character strings. Some data, such as monetary amounts and dates, is stored in a compact, internal format. For these fields, the dictionary holds a conversion code, which specifies a conversion to apply before displaying the data.

Types of dictionary records

The following main types of dictionary records define fields in the data file:

- D-descriptors, which define the data actually stored in a field
- I-descriptors, which are calculated fields, evaluated whenever the value is required
- On UniData systems, V-descriptors (which define virtual fields) are like I-descriptors

I-descriptors can perform calculations on data stored in one record, or retrieve data from other files. For example, records in the CUSTOMER file have a field that lists related record IDs in the ORDERS file. The CUSTOMER file dictionary could contain I-descriptors that use the `TRANS` function to retrieve fields from those related records.

Locks

When a program makes changes to the database, it sets a lock on each record involved in the update, ensuring that no other user or process can modify the record until the lock is released.

Locks and locking strategy are described in Record locks.

Data retrieval

UniVerse contains several utilities to use with the database, including:

- RetrieVe, a data query and reporting language
- ReVise, a menu-based data entry and modification program
- Editor, a line editor that adds, changes, and deletes records in a file

UniData provides a set of similar programs:

- UniQuery, a data query and reporting language
- UniEntry, a menu-based data entry and modification program

- Editor and AE Editor, line editors that add, change, and delete records in a file
- UniData SQL, UniData's version of the SQL language

The database also has many commands and keywords for administering and maintaining the database. All these utilities and commands can be accessed by a program through UniObjects. For more information, see Using database commands.

UniObjects concepts

If you already know UniVerse or UniData, you will find that UniObjects uses some different terms to define familiar database features. This section defines those terms and shows how they map to the database.

Objects

An object is an instance of a class. All objects of a class share characteristics. The objects you can use with UniObjects for .NET are shown in the following table. The five most commonly used classes are listed first in the order in which you are most likely to use them in an application. The remaining classes are organized in alphabetical order.

Object	Description
`UniRoot`	This is an abstract class from which all `UniObjects` for .NET classes are inherited.
`UniObjects`	Represents an open connection to the UniVerse or UniData database. This class cannot be inherited.
`UniSession`	A `UniSession` object is a reference to a connection between your client program and the database running on the server. You normally access the other objects through the `UniSession` object.
`UniFile`	A `UniFile` object is a reference to a database file.
`UniDictionary`	A `UniDictionary` object is a reference to a database file dictionary.
`UniCommand`	A `UniCommand` object is a reference to a database command executed on the server.
`UniDataSet`	A `UniDataSet` object is a reference to a set of information, such as a group of record IDs, that can be used with other objects.
`UniDynArray`	A `UniDynArray` object is a reference to a dynamic array, such as a record or select list.
`UniNLSLocale`	A `UniNLSLocale` object is a reference to the NLS locale information for a session.
`UniNLSMap`	A `UniNLSMap` object is a reference to the NLS map information for a session.
`UniSelectList`	A `UniSelectList` object is a reference to a database select list.
`UniSequentialFile`	A `UniSequentialFile` object is a reference to a type 1 or type 19 UniVerse file used for storing text, programs, or other data.

Object	Description
UniRecord	A UniRecord is a subclass of the UniDynArray class, and includes information regarding the record ID (as well as the record data) and the status of any recent operation on that data.
UniSubroutine	A UniSubroutine object is a reference to a BASIC subroutine that is called by the client program but runs on the server. For BASIC users, this is the familiar cataloged subroutine.
UniTransaction	A UniTransaction object is a reference to a transaction for a session.

Methods

Methods are procedures used with a particular object. Many of the methods used in UniObjects for .NET are equivalent to UniObjects methods and properties, and to BASIC statements and functions. For example, the ClearFile ( ) method is equivalent to the UniObjects ClearFile method and the BASIC CLEARFILE statement.

Properties

Properties represent the internal state of any given object. In UniObjects for .NET, you use a .NET property declaration. Properties are an extension of fields and are accessed using the same syntax. Unlike fields, properties have accessors that read, write, or compute their values. For example:

```
//A read-write instance property
    public string fileName
        get
            {
                return name;
            }
                set
            {
                name=value;
            }
```

Opening a database session

You must connect to a database server before you can access files or records on it. You use the OpenSession( ) method of the UniObjects object to establish a server session. The server can be the same computer that the client application is running on, or it can be a different computer linked by a network. A connected session is like any login session established by a terminal user.

Once the session is active, you can use it to create other objects. For example, if you want to open a file, execute a database command, or run a subroutine on the server, you start the operation using the methods provided by the UniSession object.

The UniSession object must exist for as long as your application needs access to the server. When a UniSession object is no longer active, this closes the connection with the database server. This means that although the objects created through a UniSession object are still available, you may not be able

to use them. For example, if you have a UniFile object, you can access the last record read from the file, but you cannot read another record.

Data encryption

Data encryption is a facility in which data transmissions between the client and server is modified to prevent unsecure parties from intercepting sensitive data. UniObjects for .NET provides the facility to use encryption at the session, object, and operation levels.

Using methods to create objects

The following table shows the UniObjects for .NET objects and the methods you use to create or access them. The methods all belong to the UniSession object.

Object	Method
UniCommand	CreateUniCommand ( )
UniDictionary	CreateUniDictionary ( )
UniDynArray	CreateUniDynArray( )
UniFile	CreateUniFile ( )
UniNLSLocale	CreateNLSLocale ( )
UniNLSMap	CreateNLSMap ( )
UniSelectList	CreateUniSelectList ( )
UniSequentialFile	CreateSequentialFile( )
UniSubroutine	CreateUniSubroutine ( )
UniTransaction	CreateUniTransaction ( )

Using the @TTY variable

During normal server operations, the @TTY variable on the server is set to the terminal number. If the process is a phantom, @TTY returns the value phantom. If the process is a database API such as UniObjects for .NET or UniObjects, @TTY returns the value `uvcs` on UniVerse systems and `udcs` on UniData systems.

You can use this returned value by adding a paragraph entry to the VOC file. For example:

```
PA
IF @TTY = 'uvcs' THEN GO END:
START.APP
END:
```

Using files

Before you can use a database file, you must open the file using the `CreateUniFile( )` method of the `UniSession` object as follows:

```
UniFile custfile = uSession.CreateUniFile("CUST");
```

Reading and writing records

When a file is open, you can read data from it and write data to it. To read a record, call the `Read ( )` method of the `UniFile` object. For example:

```
UniDynArray custRec = custFile.Read("12345");
```

To write data back to the file, call the `Write ( )` method of the `UniFile` object. For example:

```
custFile.Write();
```

Fields, values, and subvalues

When you read a record, it is returned as a `UniDynArray` object. To access or manipulate the dynamic array, you address the fields, values, and subvalues just as you do in BASIC and just as they are stored in the file.

You can address fields, values. and subvalues as follows:

```
dynArray.method (field, [ value, [ subvalue ] ] ) ;
```

`dynArray` is the object variable, method is the method you want to use, and field, value, and subvalue are integers representing the respective field, value, or subvalue you want to access. If no field is given, the operation occurs over the entire array.

For example, to find what is the third value in a dynamic array, write:

```
UniDynArray thirdField = origArray.Extract( 3 );
```

This extracts field 3 from the origArray and returns the data into the object `thirdField`.

To access a value, do the same thing, but extend it as follows:

```
UniDynArray thirdFieldSecondValue = origArray.Extract( 3, 2 );
```

This extracts the second value from the third field and returns the object into `thirdFieldSecondValue`.

To modify data in the object, do the same thing. For example, to change the second value of the third field, write:

```
origArray.Replace( 3, 2, "NewData" );
```

This changes the object immediately.

Other operations you can perform are:

- To count the number of values in field 2 of the dynamic array:

```
int NumValues = origArray.Count( 2 );
```

- To count the number of fields in the entire array:

```
int NumValues = origArray.Count();
```

- To insert a new field before field 5 in the array:

```
origArray.Insert( 5, "new value " );
```

- And finally, to delete the fourth subvalue of the first value of field 3:

```
origArray.Delete( 3, 1, 4 );
```

You can also use the other methods in the same way.

Data conversion

When you read and write an entire record, your program must handle conversion of data to and from its internal storage format. You do this using the `Iconv` ( ) (input convert) and `Oconv` ( ) (output convert) methods of the `UniSession` object.

For example:

```
UniDynArray dateBox = uSession.Oconv(x,"D");
```

In most cases the position of the field in the record and the conversion code to apply must be written into your program. This means that your program may need to change if the structure of the record changes.

As an alternative, you can read or write to a named field rather than to the entire record, and let `UniObjects` consult the file dictionary and perform any data conversion for you. You do this using the `ReadNamedField` ( ) and `WriteNamedField` ( ) methods[1].

The `ReadNamedField` ( ) method of the `UniFile` object lets a program request data in its converted form from a field specified by name. `ReadNamedField` ( ) can also evaluate I-descriptors. For example, the code to read the LAST.ORDER.DATE field might look like this:

```
UniDynArray rec = custFile.ReadNamedField('LAST.ORDER.DATE');
```

The `WriteNamedField` ( ) method does the converse, that is, it takes a data value, applies an input conversion to it, then writes it to the appropriate location in the record. It does not support I-descriptors.

[1] BASIC does not have equivalents to the ReadNamedField ( ) and WriteNamedField ( ) methods.

Error handling

UniObjects for .NET separates the code that handles error exceptions from the normal code flow. An exceptional condition is said to throw an exception that must be caught. Whenever an error occurs in one of the API libraries, the method encountering the error throws a particular exception, which the programmer must then catch and handle appropriately. This is done using .NET try/catch blocks. For example:

```
try
{
    result = uFile.Read(recordToBeRead);
```

[1] BASIC does not have equivalents to the ReadNamedField ( ) and WriteNamedField ( ) methods.

```
    result2 = uFile.Read(nextRecordToBeRead);
}
catch(Exception e)
{
    processError(e);
}
```

This ensures that normal operations are handled in one section, and exceptional conditions are handled in another section.

Many classes support a status property (for example, `FileStatus`), which enables the developer to get additional information about certain operations.

UniObjects for .NET does not have a direct equivalent to the THEN and ELSE clauses that a BASIC programmer uses to specify different actions depending on the success of an operation. Instead, all database objects throw a `UniException` object, which is set by various methods. If the method does not finish successfully, the `UniException` object indicates an error. For a list of error codes, see Error codes and replace tokens.

For example, if you call the `Read` ( ) method of a `UniFile` object, the operation fails if the record does not exist. In this case, the `UniFileException` object indicates the record was not found.

For examples of error handling, see the entry for the `UniFile` object in A tour of the objects.

Record locks

Note: BASIC programmers should read this section carefully. Locking is handled differently in UniObjects to make coding easier in the event-driven environment of a client application.

UniVerse and UniData have a system of locks to prevent potential problems when several users try to access the same data at the same time. The three types of lock you can use in programs are task locks, file locks, and record locks. This section discusses only record locks, which are used most often. For information on task locks and file locks, refer to the descriptions of the `CreateTaskLock` ( ) and `ReleaseTaskLock` ( ) methods of the `UniSession` object, and to the `LockFile` ( ) and `UnlockFile` ( ) methods of the `UniFile` object, in A tour of the objects. See also the *UniVerse System Description* for more information on record and file locks.

A record lock prevents other users from:

- Setting a file lock on the file containing the locked record.
- Setting a record lock on the locked record.
- Writing to the locked record.
- Creating a record with the same record ID. In this case, you set a lock on the record before it has been created.

There are two types of record lock:

- Exclusive update locks (READU locks), which prevent other users from reading or writing to the record
- Shared read locks (READL locks), which allow other users to read the record but not to update it

Setting and releasing locks

Setting and releasing record locks is controlled by three properties of the `UniFile` object:

- The blocking strategy set by the `UniFileBlockingStrategy` property specifies whether to wait if the record is already locked (equivalent to a BASIC LOCKED clause).
- The lock strategy set by the `UniFileLockStrategy` property specifies what kind of lock to set when reading.
- The release strategy set by the `UniFileReleaseStrategy` property specifies when to release a lock, for example:
  - When a record is written or deleted.
  - When you set a new record ID value using the RecordID property. This provides a simple way to set the lock release strategy for a program that edits a sequence of records, without having to code lock handling every time a record is read or written.
  - Only by the `UnlockRecord`() method.

Note: All locks are released when the session is closed.

You can set these properties for each file, or you can use the defaults associated with the `UniSession` object. These defaults are specified using the `BlockingStrategy`, `LockStrategy`, and `ReleaseStrategy` properties of the `UniSession` object. In either case the properties remain set for all subsequent reads on that file during the session; you do not need to set them again. For examples, see the entries for the `UniFile` and `UniSession` objects in A tour of the objects.

Select lists

In UniVerse and UniData you can retrieve a specified set of record IDs, saving them as an active select list. You can either use the active select list immediately in a program or command, or give it a name and save it for future use. A UniVerse session can have up to 11 select lists active at the same time, numbered from 0 through 10. A UniData session can have up to 10 active select lists, numbered from 0 through 9.

Accessing select lists

A `UniObjects` for .NET application can use select lists by defining `UniSelectList` objects. You get a reference to one of the numbered select lists using the `CreateSelectList`() method of the `UniSession` object. For example:

```
UniSelectList uSelect = uSession.CreateSelectList();
```

Creating select lists

The `UniSelectList` methods you can use to create a select list are `FormList` ( ), `Select` ( ), `SelectAlternateKey` ( ), or `SelectMatchingAK`( ). You can also create a select list by executing a database command that creates one; for example, `SELECT` or `SSELECT`. In the following example, the `Select`() method creates a select list:

```
uSelect.Select(uFile);
```

Reading and clearing select lists

You can read a select list in two ways:

- One record ID at a time using the `Next ( )` method
- All record IDs at once using the `ReadList ( )` method

If you just want to read part of a list, you can discard the unwanted part by calling the `ClearList()` method.

For more information and examples, see the entry for the `UniSelectList` object in A tour of the objects.

Using a dictionary

For most application programs it is economical to build a record's structure and field types into the program. This avoids having to look up the format of the record in the file dictionary. If you want your program to process different types of records, you will need to look in the file dictionary to see how the records are structured. In a UniObjects for .NET application, you do this through the `CreateUniDictionary ( )` method of the `UniSession` object. This returns a `UniDictionary` object, which has methods for reading and writing particular fields from the dictionary. These methods are:

- `GetAssoc ( )` and `SetAssoc ( )`
- `GetConv ( )` and `SetConv ( )`
- `GetFormat()` and `SetFormat()`
- `GetLoc( )` and `SetLoc ( )`
- `GetName( )` and `SetName()`
- `GetSM()` and `SetSM ( )`
- `GetSQLType()` and `SetSQLType()`
- `GetType()` and `SetType()`

For more information about these methods, see the entry for the `UniDictionary` object in A tour of the objects.

Here is an example that finds the type of a particular field:

```
UniDictionary dictFile = uSession.CreateUniDictionary() ("XXX")
UniString rec = dictFile.GetType();
```

Using binary and text files

You can use UniVerse type 1 and type 19 files to store text or binary data you want to include in a program. UniVerse implements type 1 and type 19 files as operating system directories. The records in type 1 and type 19 files are implemented as operating system files whose file names are the database record IDs:

Database item	Implemented by operating system as...
Type 1 or type 19 file	Directory
Type 1 or type 19 file record	File
Record ID	Filename

For small text files, you can open the type 1 or type 19 file with the Open ( ) method and then read an entire text file with the Read ( ) method. See Using files.

Accessing files sequentially

On UniVerse and UniData systems, if a file is large or contains binary data, it is better to read and write the file sequentially, that is, in manageable sections. You can do this by using the `CreateSequentialFile`() method of the `UniSession` object. This returns a `UniSequentialFile` object, whose methods allow sequential access to the data. The `UniSequentialFile` object uses an internal file pointer to track read and write operations (equivalent to BASIC's sequential file variable). You can:

- Read and write lines of text with the `ReadLine`() and `WriteLine`() methods
- Read and write binary data with the `ReadBlk`() and `WriteBlk`() methods
- Change the position of the file pointer with the `FileSeek`() method
- Truncate an existing file with the `WriteEOF`() method

For more information, see the entry for the `UniSequentialFile` object in A tour of the objects.

Using database commands

Your program can run most database commands through the `UniCommand` object, which is equivalent to the BASIC EXECUTE statement.

You can use the `UniCommand` object for:

- Creating or deleting a database file.
- Making a select list of records that meet your requirements. See When to use database commands.
- Running a program on the server to save processing power on the client.

> **Note:** The `UniCommand` object may not always be the most efficient way to use resources in a client/server program. For more information, see When to use database commands.

You can issue only one command at a time. You use the `CreateUniCommand` ( ) method of the `UniSession` object to create a `UniCommand` object. For example:

```
UniCommand com1 = uSession.CreateUniCommand()
```

You specify the command that you want to execute using the command property, and then execute it by calling the `Execute`( ) method. For example:

```
com1.command="some command";
com1.Execute();
```

You can get the result of a command using the `CommandStatus` property and Response property as follows:

- If the command ran to completion, the `CommandStatus` property returns `UniObjectsTokens.UVS_COMPLETE`, and you can get any output generated by the command using the `Response` ( ) method.
- If the command did not finish, or if all the output was not retrieved, the `CommandStatus` property shows what happened. You can use the `Reply` ( ) or `NextBlock` ( ) method to continue processing. For an example, see the entry for the `UniCommand` object in A tour of the objects.

Client/server design considerations

In designing your application, avoid unnecessary interaction between the client and the server. This has two main benefits:

- Performance: reducing network traffic improves performance.
- Scalability: if more clients and servers are added to the network, your application's performance remains acceptable.

To use the client and server efficiently, you must know which operations need to communicate with the server and when those operations take place. If necessary, you can then change the design of the application to reduce the interaction with the server. The following sections describe some ideas for using the client and server economically.

Calling server subroutines

You can reduce network traffic by running parts of your application on the server as BASIC subroutines. Server subroutines run in an area called catalog space that is available to any program on the server.

Note: A server subroutine must be cataloged before you can call it from UniObjects. For more information about cataloging subroutines on UniVerse systems, see the entry for the `CATALOG` command in *UniVerse User Reference*, and the discussion of subroutines in *UniVerse BASIC*. For information about cataloging UniData subroutines, see *UniData Commands Reference* and *Administering UniData*.

Your program can call a cataloged subroutine via the UniSubroutine object, which you get using the CreateUniSubroutine() method of the UniSession object. For example:

```
UniSubroutine getOrderData = uSession.CreateUniSubroutine()
                        ("*GET.ORDER.DATA", 4);
```

The `CreateUniSubroutine`() method needs the name of the cataloged subroutine and the number of arguments that it takes. Once your program has obtained the `UniSubroutine` object, you use the `SetArg` ( ) method to supply values for arguments, the `Call`( ) method to call the subroutine, and the `GetArg` ( ) method to retrieve any argument values returned. For example:

```
getOrderData.SetArg(0, OrderNumber);
getOrderData.SetArg(1, DisplayType);
getOrderData.Call();
UniString displayValue = getOrderData.GetArg(2);
```

When to use database commands

You can save client processing by executing database commands on the server. The most effective commands to use are those that do not generate any output, such as the RetrieVe and UniQuery `SELECT` and `SSELECT` commands.

Some commands can increase network traffic because they generate prompts or messages that your program must then handle. If your program cannot cope with an unexpected request for input from a command, it hangs, with no indication of what went wrong. In particular, avoid using interactive commands such as `CREATE.FILE` or `REFORMAT` which have many possible prompts and error conditions. (In most cases it should not be necessary to create or reformat files as part of your application.)

Task locks

You can protect a process running on the server from interruption by other users or programs by setting a task lock. UniVerse and UniData have 64 task locks you can assign to events or processes. For example, if your application uses a resource such as a printer, you can set a task lock to prevent another database user from accessing the printer during your print run.

You set and release task locks with the `SetTaskLock`() and `ReleaseTaskLock`() methods of the `UniSession` object. Task locks have no predefined meanings. You must ensure that your application sets and releases task locks efficiently. You can use the `LIST.LOCKS` command to check which locks are in use and which users hold them.

Connection pooling

Connection pooling maintains connections between a client and a server. Instead of ending a connection when it is not needed and establishing a new connection when one is required, connections are returned to a pool and made available for other requests, improving overall system performance.

The connection pooling process is outlined below:

1. Ensure connection pooling is activated. Refer to Activating connection pooling for instructions.
2. When a server receives a connection request and opens its first session, it initializes the connection pool based on the pool configuration settings. Once the connection pool is established, it will remain open and available for the next request and cannot be changed unless it is explicitly deactivated.

   Each connection pool is based on the same credential parameters:

   - The server name
   - The account name
   - The ID and password used
3. When an additional connection request is received, the connection pool will check to see if a session is available.
   - If a session is available, the request will use the existing connection to process the request.
   - If a session is not available, the pool will create another connection until the maximum connection pool size is met.

The minimum and maximum number of connections a pool can contain are specified in the pool configuration settings. Each connection consumes one connection pool license.

- If the maximum pool size is met, the request will reside in the queue until the pool can create another connection, or until the request is released or times out.

4. When the request is complete, the process returns the result set and makes the connection available for a new request.

Note: Connection pooling does not maintain connections when they are not used. As a result, when a process ends, the associated application might not be aware of the connection being lost and attempt to reuse a process that no longer exists. (Connections are most often lost when servers are restarted, UniRPC services are restarted, or when processes unexpectedly end.) If a connection loss occurs, the process will not be reused and will generate a new connection request instead. Consequently, if an application fails to open, try to use it once again and the session will likely be re-established.

The overall connection pool response time consists of:

- Time to establish the connection.
- Application initialization, which might include opening files or setting up an environment.
- Application logic and performance.
- Returning results, which often depends on the size of the result set. Larger result sets take longer to return.

Connection pool size

Connection pool connections dynamically change, based on system demand, between the minimum and maximum sizes specified. When no connections are available, the server either creates another connection if the maximum connection pool size has not yet been reached, or keeps the thread waiting in the queue until a connection is released or times out. If a pooled connection is idle for a specified time, it is disconnected.

License considerations

The actual size of a connection pool depends on the pooling licenses available on the server. For example, if you set a connection pool to a minimum size of 2 and a maximum size of 100, and you have 16 licenses available, the maximum connection pool size will be 16. If you only have 1 license available, a connection pool is not initialized at all, since the minimum size of 2 cannot be met.

Connection allocation

Once a pooled connection is allocated to a thread, the connection remains exclusively attached to that thread until it is explicitly freed by the thread.

Pooled connections are not "cleaned up" before they are allocated to another thread with the same credentials. For example, **UDT.OPTIONS** settings and unnamed common and environment variables remain from previous use.

Activating connection pooling

To activate connection pooling, use the **UniObjects.UOPooling** statement in your program, as shown in the following example:

```
UniObjects.UOPooling = true;
```

Specifying the size of the connection pool

To specify the size of the connection pool, use **UniObjects.MinPoolSize** to define the minimum number of connections, and the **UniObjects.MaxPoolSize** to define the maximum number of connections, as shown in the following example:

```
UniObjects.MinPoolSize = 1;
UniObjects.MaxPoolSize = 10;
```

If you do not specify the minimum and maximum number of connections, the system defaults to `1` for the minimum and `10` for the maximum.

Creating multiple connection pools

You can create as many connection pools as you want by issuing multiple **UniObjects.OpenSession** commands in your program. Each connection pool has the exact same server name, login name, password and account. A new connection pool is created if any of these parameters change, except the *service_name* parameter.

`UniObjects.OpenSession``(`*`server_name, logon_name, password,account,service_name`*`);`

The following table describes each parameter of the syntax.

Parameter	Description
server_name	The name of the server to which you are connecting.
logon_name	The logon name of the user connecting to the server.
password	The password corresponding to the logon_name.
account	The account on the server to which you are connecting.
service_name	This parameter is optional. The name of the RPC service. If you do not specify *service_name*, the default is **defcs**. If you do specify *service_name*, the service name must exist in the `unirpcservices` file.

Connection pooling code example

The following example illustrates using connection pooling in a program:

```
using System;
using U2.UODOTNET;
using System.Threading;

namespace CPTest
{
        class CPTest
        {
```

```
            [STAThread]
            static void Main(string[] args)
        {
                UniSession us1=null;
                try
                {
                    UniObjects.UOPooling = true;
                    UniObjects.MinPoolSize = 1;
                    UniObjects.MaxPoolSize = 10;
                    us1 =
UniObjects.OpenSession("localhost","username","password","demo","udcs");
                    UniCommand cmd = us1.CreateUniCommand();
                    cmd.Command="SSELECT STATES";
                    cmd.Execute();
                    string response_str = cmd.Response;
                    Console.WriteLine(" Response from UniCommand
:"+response_str);
                    UniSelectList sl = us1.CreateUniSelectList(0);
                    while (!sl.LastRecordRead)
                    {
                        string s = sl.Next();
                        if (s != "")
                        {
                            Console.WriteLine(" Record ID : "+s);
                        }
                    }
                }
                catch(Exception e)
                {
                    if(us1 != null && us1.IsActive)
                    {
                        UniObjects.CloseSession(us1);
                        us1= null;
                    }
                    Console.WriteLine("");
                        Console.WriteLine(Thread.CurrentThread.Name +" :
Connection failed in Test Program : " + e.Message +e.StackTrace);
Console.WriteLine("============================================================
======================");
Console.WriteLine("============================================================

====================");
                                                }
                (finally
                {
                    if(us1 != null && us1.IsActive)
                    {
                    Console.WriteLine("");
                    Console.WriteLine(Thread.CurrentThread.Name +" :
Connection Passed in Test Program");
Console.WriteLine("============================================================
====================");
Console.WriteLine("============================================================
====================");
                                    UniObjects.CloseSession(us1);
                    }
                }
            }
        }
}
```

Configuration file example

The following example illustrates a configuration file for connection pooling. This configuration file is named either app.config or web.config.

```
<?xml version="1.0" encoding="utf-8" ?>
- <configuration>
    - <UO.NET>
        - <General>
            <add key="SocketTimeOut" value="300000" />
        </General>
    - <ConnectionPooling>
    <add key="ConnectionPoolingOn" value="1" />
    <add key="MinimumPoolSize" value="1" />
    <add key="MaximumPoolSize" value="16" />
    <add key="IdleRemoveThreshold" value="300000" />
    <add key="IdleRemoveExecInterval" value="6000" />
    <add key="OpenSessionTimeout" value="30000" />
        </ConnectionPooling>
    - <PerformanceMonitor>
    <add key="BusyConnectionCounter" value="0" />
        </PerformanceMonitor>
    </UO.NET>
    -
-<system.diagnostics>
    + <switches>
    - <trace autoflush="true" indentsize="4">
        - <listeners>
     <add name="myListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="c:\temp\myListener.log" />
            </listeners>
        </trace>
    </system.diagnostics>
</configuration>
```

The following table describes the configuration parameters for connection pooling:

Parameter	Description
ConnectionPoolingOn	When this value is set to 1, the connection is drawn from the appropriate pool, or, if necessary, created and added to the appropriate pool. The default value is 0.
MinimumPoolSize	The minimum number of connections maintained in the connection pool.
MaximumPoolSize	The maximum number of connections in the connection pool.
IdleRemoveThreshold	Determines the amount of time, in milliseconds, one session can remain idle in the connection pool.
IdleRemoveExecInterval	The thread execution interval time, in milliseconds. During this interval, UniObjects for .NET removes idle sessions in the connection pool.
OpenSessionTimeOut	How much time UniObjects for .NET waits before timing out to get a session from the connection pool. Expressed in milliseconds.

Using encryption wallets

You can create an encryption key wallet, which contains encryption keys and passwords. Instead of activating encrypt keys individually, you can supply a wallet and its corresponding password to UniData or UniVerse to activate all the encryption keys contained in the wallet. UniData and UniVerse store the wallet in the key store.

Client applications can use the encryption key wallet to activate keys for the entire session. Call the ACTIVATE_WALLET() and DEACTIVATE_WALLET() subroutines to activate and deactivate encryption keys contained in the wallet. Each of these subroutines have the following parameters, which must be supplied in the following order:

1. Wallet_id – The ID of the encryption key wallet
2. Wallet_password – The password for the encryption key wallet
3. Status – 0 for success, other codes indicate failure
4. Error_message – In case of failure, a detailed error message

For clients that do not use SQL to access the database, you can call these procedures directly from a session object.

If you are using UniObjects, you can access these methods through the subroutine method of a session object.

For more information on automatic data encryption, see the *UniVerse Security Features* manual.

Chapter 3: A tour of the objects

This chapter describes the classes used in UniObjects for .NET and details their associated constructors, properties, and methods. The five most commonly used classes are listed first in the order in which you are most likely to use them in an application. The remaining classes are organized in alphabetical order.

Object	Description
UniRoot	`UniRoot` is an abstract class from which all UniObjects for .NET classes are inherited.
UniObjects	The `UniObjects` class represents an open connection to the UniData or UniVerse database.
UniSession	The `UniSession` class defines and manages a database session on the server. It controls access to all database objects dependent on it.
UniFile UniDictionary	Next, your program is likely to access a database file on the server through a `UniFile` or `UniDictionary` object.
UniCommand	The `UniCommand` class controls execution of database commands on the server.
UniDataSet	`UniDataSet` is a collection class. It provides a collection interface for sets of `UniRecord` objects, which can then be used to perform bulk or batch operations with one network operation.
UniDynArray	You can address records in database files through the `UniDynArray` class. You can also use this object independently of a session.
UniNLSLocale	On UniVerse systems, the `UniNLSLocale` class defines and manages the five National Language Support conventions: Time, Numeric, Monetary, Ctype, and Collate.
UniNLSMap	On UniVerse systems, the `UniNLSMap` class controls NLS map settings.
UniRecord	The `UniRecord` class controls database record interaction.
UniSelectList	The `UniSelectList` object enables you to manipulate a select list on the server.
UniSequentialFile	If you want to use data in an operating system file, you use the `UniSequentialFile` class.
UniSubroutine	The `UniSubroutine` class allows you to run a cataloged BASIC subroutine on the server.
UniTransaction	The `UniTransaction` class provides methods to start, commit, and roll back transactions for a session.
UniXML	The `UniXML` class provides methods to create XML documents and XML Schema documents from UniQuery or UniData SQL, or directly from a data file.

Note that exception classes are not documented in this chapter. For information on exception classes, please see UniObjects for .NET Help.

Code examples

The following objects include a short program example that illustrates many of the methods associated with the object: `UniObjects`, `UniSession`, `UniFile`, `UniDictionary`, `UniSequentialFile`, `UniDynArray`, `UniSelectList`, `UniCommand`, and `UniSubroutine`.

The names of the objects, constructors, properties, and methods used in UniObjects for .NET are case-sensitive.

Database account flavors

On UniVerse systems, UniObjects for .NET works best in IDEAL flavor UniVerse accounts. In other account flavors, status or error codes returned by some methods may vary from those documented.

On UniData systems, ECLTYPE U is best. You may encounter variations with other ECLTYPE or UDT.OPTIONS settings.

Constructors, properties, and methods quick reference

The following tables are a quick reference to the constructors, properties, and methods available with each object.

UniRoot constructors, properties, and methods

The following table lists the `UniRoot` constructors, properties, and methods.

Public static properties	Public instance constructors	Public instance methods	Protected instance methods
RPCDUMP	UniRoot	Dispose()	Dispose()
		Equals()	Finalize()
		GetHashCode()	MemberwiseClone()
		GetType()	
		ToString()	

UniObjects methods

The following table lists the `UniObjects` properties and methods.

Public static properties	Public static methods	Public instance methods
EnableServerAlive	CloseSession()	Dispose()
IdleRemoveExecInterval	OpenSession()	Equals()
IdleRemoveThreshold	OpenSecureSession	GetHashCode()
MaxPoolSize		GetType()
MinPoolSize		ToString()
PoolingOpenSessionTimeOut		
SslCheckCertificateRevocation		
SslIgnoreCertificateNameMismatch		
SslIgnoreIncompleteCertificateChain		
TimeOut		
UOPooling		
UOReserved1		
UOReserved2		
LastServerError		

UniSession properties and methods

The following table lists the `UniSession` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
Account	ByteArrayToUniCodeString()	
BlockingStrategy	Clone()	
CurrentOpenFiles	CreateSequentialFile()	
DeviceName	CreateTaskLock()	
EncryptionEnabled	CreateUniCommand()	
EncryptionType	CreateUniDataSet()	
HostName	CreateUniDictionary()	
HostPort	CreateUniDynArray()	
HostType	CreateUniFile()	
IPAddress	CreateUniNLSLocale()	
IsActive	CreateUniNLSMap()	
IsDisposed	CreateUniSelectList()	
LockStrategy	CreateUniSubroutine()	
MacAddress	CreateUniTransaction()	
MaxOpenFiles	Dispose()	
NLSEnabled	Encrypt()	
NLSLocalesEnabled	Equals()	
	GetAtVariable()	
	GetDelimitedByteArrayRecordID()	
Password	GetDelimitedString()	
ReleaseStrategy	GetHashCode()	
ServerVersion	GetMarkCharacter()	
Service	GetType()	
Timeout	Iconv()	
TransportType	Oconv()	
	ReleaseTaskLock()	
UOEncoding	SetAtVariable()	
UserName	ToString()	
	UniCodeStringToByteArray()	

UniFile properties and methods

The following table lists the `UniFile` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
EncryptionType	ClearFile()	Dispose()
FileName	Close()	Finalize()
FileStatus	DeleteRecord()	MemberwiseClone()
FileType	Dispose()	
IsFileOpen	Equals()	
Record	GetAkInfo()	
RecordID	GetHashCode()	
RecordString	GetType()	
UniFileBlockingStrategy	IsRecordLocked()	
UniFileLockStrategy	iType()	
UniFileReleaseStrategy	LockFile()	
	LockRecord()	
	Open()	
	Read()	
	ReadField()	
	ReadFields()	
	ReadNamedField()	
	ReadNamedFields()	
	Read Records()	
	ToString()	
	UnlockFile()	
	UnlockRecord()	
	Write()	
	WriteField()	
	WriteFields()	
	WriteNamedField()	
	WriteNamed Fields()	
	Write Records()	

UniDictionary properties and methods

The following table lists the `UniDictionary` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
EncryptionType	ClearFile()	Dispose()
FileName	Close()	Finalize()
FileStatus	DeleteRecord()	MemberwiseClone()
FileType	Dispose()	
IsFileOpen	Equals()	
Record	GetAkInfo()	
RecordID	GetAssoc()	
RecordString	GetConv()	
UniFileBlocking	GetFormat()	
Strategy	GetHashCode()	
UniFileLockStrategy	GetLoc()	
UniFileReleaseStrategy	GetName()	
	GetSM()	
	GetSQLType()	
	GetType()	
	IsRecordLocked()	
	iType()	
	LockFile()	
	LockRecord()	
	Open()	
	Read()	
	ReadField()	
	ReadFields()	
	ReadNamedField()	
	ReadNamedFields()	
	ReadRecords()	
	SetAssoc()	
	SetConv()	
	SetFormat()	
	SetLoc()	
	SetName()	
	SetSM()	
	SetSQLType()	
	SetType()	
	ToString()	
	UnlockFile()	

Public instance properties	Public instance methods	Protected instance methods
	UnlockRecord()	
	Write()	
	WriteField()	
	WriteFields()	
	WriteNamedField()	
	WriteNamedFields()	
	WriteRecords()	

UniCommand properties and methods

The following table lists the `UniCommand` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
Command	Cancel()	Dispose()
CommandAtSelected	Dispose()	Finalize()
CommandBlockSize	Equals()	MemberwiseClone()
CommandStatus	Execute()	
EncryptionType	GetHashCode()	
Response	GetType()	
SystemReturnCode	NextBlock()	
	Reply()	
	ToString()	

UniDataSet constructors, properties, and methods

The following table lists the `UniDataSet` constructors, properties, and methods.

Public instance constructors	Public instance properties	Public instance methods	Protected instance methods
UniDataSet	AfterLast	Absolute()	Dispose()
	BeforeLast	Add()	Finalize()
	CurrentRow	Clear()	MemberwiseClone()
	DelimitedByteArrayRecord	Dispose()	
	DelimitedByteArrayRecordID	Equals()	
	First	GetEnumerator()	
	Item	GetHashCode()	
	Last	GetRecord()	
	RowCount	GetRecordStatus()	
		GetType()	
		Insert()	
		Relative()	
		Remove()	
		ToString()	

UniDynArray constructors, properties, and methods

The following table lists the `UniDynArray` constructors, properties, and methods.

Public instance constructors	Public instance properties	Public instance methods	Protected instance methods
UniDynArray	StringValue	Count()	Dispose()
		Dcount()	Finalize()
		Delete()	MemberwiseClone()
		Dispose()	
		Equals()	
		Extract()	
		GetHashCode()	
		GetType()	
		Insert()	
		Length()	
		PrintByteArray()	
		Remove()	
		Replace()	
		ToByteArray()	
		ToString()	

UniNLSLocale properties and methods

The following table lists the `UniNLSLocale` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
ClientNames	Dispose()	Dispose()
ServerNames	Equals()	Finalize()
	GetHashCode()	MemberwiseClone()
	GetType()	
	SetLocaleName()	
	ToString()	

UniNLSMap properties and methods

The following table lists the `UniNLSMap` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
ServerMapName	Dispose()	Dispose()
UniMarks	Equals()	Finalize()
	GetClientMapName()	MemberwiseClone()
	GetHashCode()	
	GetType()	
	SetClientMapName()	
	ToString()	

UniRecord constructors, properties, and methods

The following table lists the `UniRecord` constructors, properties, and methods.

Public instance constructors	Public instance properties	Public instance methods	Protected instance methods
UniRecord	Record	Dispose()	Dispose
	RecordID	Equals()	Finalize
	RecordReturnValue	GetHashCode()	MemberwiseClone()
	RecordStatus	GetType()	
		ToString()	

UniSelectList properties and methods

The following table lists the `UniSelectList` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
LastRecordRead	ClearList()	Dispose()
	Dispose()	Finalize()
	Equals()	MemberwiseClone()
	FormList()	
	GetHashCode()	
	GetList()	
	GetType()	
	Next()	
	ReadList()	
	SaveList()	
	Select()	
	SelectAlternateKey()	
	SelectMatchingAK()	
	ToString()	

UniSequentialFile properties and methods

The following table lists the `UniSequentialFile` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
EncryptionType	Close()	Dispose()
IsFileOpen	Dispose()	Finalize()
ReadSize	Equals()	MemberwiseClone()
TimeOut	FileSeek()	
UniSequentialStatus	GetHashCode()	
	GetType()	
	Open()	
	ReadBlk()	
	ReadLine()	
	ToString()	
	WriteBlk()	
	WriteEOF()	
	WriteLine()	

UniSubroutine properties and methods

The following table lists the `UniSubroutine` properties and methods.

Public instance properties	Public instance methods	Protected instance methods
ArgumentsNumber	Call()	Dispose()
RoutineName	Dispose()	Finalize()
	Equals()	MemberwiseClone()
	Get Arg()	
	GetArgDynArray()	
	GetHashCod()e	
	GetType()	
	ResetArgs()	
	SetArg()	
	ToString()	

UniTransaction methods

The following table lists the `UniTransaction` methods.

Public instance methods	Protected instance methods
Begin()	Dispose()
Commit()	Finalize()
Dispose()	MemberwiseClone()
Equals()	
GetHashCode()	
GetLevel()	
GetType()	
IsActive()	
Rollback()	
ToString()	

UniObjects and BASIC equivalents

The following table shows the UniObjects for .NET methods and properties and their equivalents in UniObjects and BASIC.

Method/property	UniObjects equivalent	BASIC equivalent
`Call ()`	Call	`CALL`
`Cancel ()`	Cancel	No direct equivalent
`ClearFile ()`	ClearFile	`CLEARFILE`
`ClearList ()`	ClearList	`CLEARSELECT`
`Close ()`	CloseFile	`CLOSE`, reassignment to file variable
	CloseSeqFile	`CLOSESEQ`
`CloseSession()`	Disconnect	No direct equivalent

Method/property	UniObjects equivalent	BASIC equivalent
Commit()	Commit	COMMIT (UniVerse) TRANSACTION COMMIT (UniData)
Count ()	Count	DCOUNT( )
CreateUniFile()	OpenFile OpenDictionary OpenSequential	OPEN OPEN DICT OPENSEQ
DeleteRecord()	DeleteRecord	DELETE, DELETEU
Execute()	Exec	EXECUTE
FileSeek()	FileSeek	SEEK
FormList()	FormList	FORMLIST
GetAkInfo()	GetAkInfo	INDICES( )
GetArg()	GetArg	No direct equivalent
GetAtVariable()	GetAtVariable	No direct equivalent
GetList()	GetList	No direct equivalent
Iconv()	Iconv	ICONV( )
IsActive()	IsActive	No direct equivalent
IsFileOpen	IsOpen	No direct equivalent
iType()	IType	ITYPE( )
Length()	Length	No direct equivalent
LockFile()	LockFile	FILELOCK
LockRecord()	LockRecord	RECORDLOCKL RECORDLOCKU
Next()	Next	READNEXT
NextBlock()	NextBlock	No direct equivalent
Oconv()	Oconv	OCONV( )
OpenSession()	Connect	No direct equivalent
Read()	Read	READ, READL, READU
ReadBlk()	ReadBlk	READBLK
ReadField()	ReadField	READV, READVL, READVU
ReadLine()	ReadLine	READSEQ
ReadList()	ReadList	READLIST
ReadNamedField()	ReadNamedField	No direct equivalent.
ReleaseTaskLock()	ReleaseTaskLock	UNLOCK
Replace()	Replace	REPLACE( )
Reply()	Reply	No direct equivalent
ResetArgs()	ResetArgs	No direct equivalent
Rollback()	Rollback	ROLLBACK (UniVerse) TRANSACTION ABORT (UniData)
SaveList()	SaveList	No direct equivalent
Select()	Select	SELECT
SelectAlternateKey()	SelectAlternateKey	SELECTINDEX
SelectList()	SelectList	No direct equivalent

Method/property	UniObjects equivalent	BASIC equivalent
SelectMatchingAK()	SelectMatchingAk	SELECTINDEX
SetArg()	SetArg	No direct equivalent
SetAtVariable()	SetAtVariable	No direct equivalent
SetName()	SetName	No direct equivalent
SetTaskLock()	SetTaskLock	LOCK
Start()	Start	BEGIN TRANSACTION (UniVerse) TRANSACTION START (UniData)
Subroutine()	Subroutine	No direct equivalent
SubValue()	SubValue	No direct equivalent
UnlockFile()	UnlockFile	FILEUNLOCK
UnlockRecord()	UnlockRecord	RELEASE
Value()	Value	No direct equivalent
Write()	Write	WRITE, WRITEU
WriteBlk()	WriteBlk	WRITEBLK
WriteEOF()	WriteEOF	WEOFSEQ
WriteField()	WriteField	WRITEV, WRITEVU
WriteLine()	WriteLine	WRITESEQ
WriteNamedField()	WriteNamedField	No direct equivalent

UniRoot class

The UniRoot class is an abstract class. All UniObjects for .NET classes are inherited from the `UniRoot` class. Tracing functionality is implemented in this class.

UniRoot – public static properties

This section describes public static properties you can use with `UniRoot` objects.

public static int RPCDUMP {get; set;}

This property gets or sets the RPC dump level.

UniRoot – public instance constructors

This section describes the public instance constructor for the `UniRoot` class.

UniRoot( )

This is the default constructor for the class. The constructor takes no arguments.

UniRoot – public instance methods

This section describes the public instance methods you can use with `UniRoot` objects.

public void Dispose( )

This method performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniRoot – protected instance methods

This section describes the protected instance methods you can use with `UniRoot` objects.

protected override void Dispose (bool disposing)

This method overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniObjects class

The `UniObjects` class represents an open connection to the UniData or UniVerse database. This class cannot be inherited.

UniObjects – public static methods

This section describes the public static methods you can use with `UniObjects` objects.

public static LastServerError

This method reports back the last error reported to the server , including missing subroutine calls, unassigned variables etc.

The following example is using the property in Visual Basic .NET with UniObject .NET:

```
LastSeverErrorMess = MachineASession.LastServerError
If LastSeverErrorMess <> "" Then
MsgBox("Subroutine encountered the following run time error " +
  LastSeverErrorMess, MsgBoxStyle.Information)
End If
```

public static void CloseSession (UniSession us)

This method closes the connection to the UniData or UniVerse database, closes any open files, and releases any locks associated with the session.

`us` is the UniSession object to be closed.

After calling this method, the `UniSession.IsActive` property returns false, and any operation performed on this session, other than `OpenSession( )` or `CloseSession ( )`, results in an error and throws an exception.

Note: Other objects created by or associated with the session are still available, but using them may cause an error. For example, if you have aUniFile object created by the `UniSession` object, you can access the last record that was read from the file, but you cannot read another record.

If this method fails, it throws an `Exception`.

This method corresponds to the `UniObjects Disconnect` method.

The following example closes the connection to the database:

```
if (us !=null && us.IsActive)
{
    UniObjects.CloseSession(us);
}
```

public static UniSession OpenSession (string hostname, string userid, string password, string account)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

public static UniSession OpenSession (string hostname, string userid, string password, string account, string service)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`string service` is the type of UniData or UniVerse database account: `udvs` for UniData or `uvcs` for UniVerse.

public static UniSessession OpenSession(string hostname, int port, string userid, string password, string account)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`int port` is the port number on the host to use for the connection.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

public static UniSessession OpenSession(string hostname, int port, string userid, string password, string account, string service)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`int port` is the port number on the host to use for the connection.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`string service` is the type of UniData or UniVerse database account: `udvs` for UniData or `uvcs` for UniVerse.

If this group of method fails, it throws an `Exception`.

```
UniSession us=null;
                try
                {
us = UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");
```

```
UniCommand runCmd = uSession.CreateUniCommand( ) ;
runCmd.Command = "RUN BP FOO" ;
runCmd.Execute();

}
```

```
catch(Exception e)
{
    Console.WriteLine(e.Message +e.StackTrace);
}
finally
{
    if(us != null && us.IsActive)
    {
        UniObjects.CloseSession(us);
    }
}
```

public static UniSession OpenSecureSession (string hostname, string userid, string password, string account, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`X509Certicate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null.

public static UniSession OpenSecureSession (string hostname, string userid, string password, string account, string service, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname`is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`string service` is the type of UniData or UniVerse database account: udvs for UniData or uvcs for UniVerse.

`X509Certicate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null.

public static UniSession OpenSecureSession (string hostname, int port, string userid, string password, string account, X509Certificate clientcertificate))

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`int port` is the port number on the host to use for the connection.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`X509Certicate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null.

public static UniSession OpenSecureSession (string hostname, int port, string userid, string password, string account, string service, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`int port` is the port number on the host to use for the connection.f

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`string service` is the type of UniData or UniVerse database account: `udvs` for UniData or `uvcs` for UniVerse.

`X509Certicate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null.

If any of this group of methods fail, it throws an `Exception`.

UniObjects – public instance methods

This section lists the public instance methods you can use with UniObjects objects.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniObjects properties

This section describes the public instance properties you can use UniObjects objects.

public static bool EnableServerAlive { get; set; }

This property gets or sets the flag that indicates whether to check if the server-end of a pooled connection is still alive before giving it to the application.

public static int IdleRemoveExecInterval {get; set;}

This property gets or sets the thread execution interval time. During this interval, idle sessions in the connection pool are removed.

public static int IdleRemoveThreshold {get; set;}

This property gets or sets the amount of time, in milliseconds, that one session can remain idle in the connection pool.

public static int MaxPoolSize {get; set;}

This property gets or sets the default maximum connection pool size.

public static int MinPoolSize {get;set;}

This property gets or sets the default minimum connection pool size.

public static int PoolingOpenSessionTimeOut {get; set;}

This property gets or sets how long UniObjects for .NET waits before timing out to get a session from the connection pool.

public static bool SslIgnoreCertificateNameMismatch {get; set;}

This property gets or sets the flag that indicates whether to ignore name mismatch errors on the server certificate during authentication. Name mismatch errors occur when the name specified in the certificate is different from the name of the server machine which provides the certificate.

public static bool SslCheckCertificateRevocation {get; set;}

This property gets or sets the flag that indicates whether the certificate revocation list is checked during authentication.

public static bool SslIgnoreIncompleteCertificateChain {get; set;}

This property gets or sets the flag that indicates whether an incomplete chain error is ignored during authentication.

public static int TimeOut {get; set;}

This property gets or sets the length of time before the operation times out. The remote procedure call (UniRPC) utility uses the timeout setting.

public static bool UOPooling {gets; set;}

This property gets or sets the connection pooling flag.

public static bool UseIPv6 {gets; set}

This property gets or sets the IPv6 flag .

public static object UOReserved1 {set;}

This property is for internal U2 processes only. Do not use this property.

public static object UOReserved2 {set;}

This property is for internal U2 processes only. Do not use this property.

UniSession class

The `UniSession` object defines and manages a database session on the server. It is the central object for any database session, controlling access to all objects dependent on it.

UniSession – public instance properties

This section describes the public instance properties you can use with `UniSession` objects.

public string Account {get;}

This property returns the account path, which is the name of the database account to which the session is connected.

This property corresponds to the UniObjects **AccountPath** property.

public int BlockingStrategy {get; set;}

This property gets or sets the default blocking strategy used in the session for all `UniFile` and `UniDictionary` objects.

`int` is the token number for the blocking strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.UVT_WAIT_LOCKED	If the record is locked, wait until it is released.
2	UniObjectsTokens.UVT_RETURN_LOCKED	Return a status value indicating the state of the lock. This is the default value.

Changing the blocking strategy does not affect existing `UniFile` or `UniDictionary` objects.

If this property fails, it throws a `UniSessionException`.

This property corresponds to the UniObjects **DefaultBlockingStrategy** property.

public int CurrentOpenFiles {get;}

This property returns the number of files that are currently open.

public bool EncryptionEnabled {get;}

This property returns `true` if encryption is enabled for this connection, otherwise false.

Note: Encryption is not available when connecting to servers running a version of UniVerse earlier than Release 9.5.

public int EncryptionType {get; set}

This property gets or sets the default encryption type for the session.

`int` is the token number for the encryption type, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_ENCRYPT	No encryption. This is the default value.
1	UniObjectsTokens.UV_ENCRYPT	Encrypt all data using internal database encryption.

If you set UV_ENCRYPT for a session, all data transferred between client and server is encrypted.

If this property fails, it throws a `UniSessionException`.

public string HostName {get;}

This property returns the name of the database server as specified by the `HostName` property.

It corresponds to the UniObjects **HostName** property.

public int HostPort {get;}

This property returns the port number on the host to use for the connection.

public int HostType {get;}

This property returns the type of host on which the UniData or UniVerse database server is running.

`int` is the token number for the host type, as follows:

Token number	Token	Description
0	UniObjectsTokens.UVT_NONE	The host system cannot be determined; the session is not connected.
1	UniObjectsTokens.UVT_UNIX	The host is a UNIX system.
2	UniObjectsTokens.UVT_NT	The host is a Windows system.

This property corresponds to the UniObjects **HostType** property.

public int IPAddress {get;}

This property returns the internet protocol (IP) address.

If this property fails, it throws a `UniSessionException`.

public bool IsActive {get; set;}

This property gets or sets the value that indicates whether the session is active.

It corresponds to the UniObjects **IsActive** method.

public bool IsDisposed {get; set;}

This property gets or sets the value that indicates whether the session is disposed.

public bool IsSecure {get;}

This property gets the SSL flag of the session.

public int LockStrategy {get; set;}

This property gets or sets the default locking strategy used in this session for all `UniFile` and `UniDictionary` objects.

`int` is the token number for the locking strategy, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_LOCKS	No locking. This is the default value.
1	UniObjectsTokens.UVT_EXCLUSIVE_READ	Sets an exclusive update lock (READU).
2	UniObjectsTokens.UVT_SHARED_READ	Sets a shared read lock (READL).

Altering the lock strategy does not affect files or dictionaries that are already open.

If this property fails, it throws a `UniSessionException`.

This property corresponds to the UniObjects **DefaultLockStrategy** property.

public string MacAddress {get;}

This property returns the value of the MacAddress data member.

public int MaxOpenFiles {get;}

This property returns the maximum number of files that can be open at the same time.

public bool NLSEnabled {get;}

This property gets the value of the NLS Map flag, which indicates whether the NLS map for the UniData or UniVerse database is enabled. Returns true if the database connection is NLS-ready, or `false` if the connection does not support NLS.

public bool NLSLocalesEnabled {get;}

This property gets the value of the NLS Locales flag, which indicates whether NLS locales are enabled for the UniData or UniVerse database. Returns true if both NLS and NLS locales are enabled for the UniData or UniVerse database, or `false` if they are not enabled.

public string Password {get;}

This property returns the password for the specified user.

It corresponds to the UniObjects **Password** property.

public int ReleaseStrategy {get; set;}

This property gets or sets the default release strategy used in the session for all `UniFile` and `UniDictionary` objects. Whenever the record ID is reset with the `RecordID` ( ) property, the release strategy reverts to the initial value.

Altering the release strategy does not affect files or dictionaries that are already opened.

`int` is the token number for the release strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.WRITE_RELEASE	Releases the lock when the record is written. This is the property's initial value.
2	UniObjectsTokens.UVT_READ_RELEASE	Releases the lock when the record is read.
4	UniObjectsTokens.UVT_EXPLICIT_RELEASE	Maintains locks as specified by the `UniFileLockStrategy` ( ) property. You can release locks only with the `UnlockRecord` ( ) method.
8	UniObjectsTokens.UVT_CHANGE_RELEASE	Releases the lock whenever a new value is set by the `RecordId`( ) property.

All the values are additive. If you specify EXPLICIT_RELEASE with WRITE_RELEASE and READ_RELEASE, it takes a lower priority. The initial release strategy value is 12, that is, release locks when the record ID changes or when locks are released explicitly.

If this property fails, it throws a `UniSessionException`.

This property corresponds to the UniObjects **ReleaseStrategy** property.

public int ServerVersion {get;}

This property returns the version of the backend UniData or UniVerse database server. A value of less than 2 represents a pre-Release 9.5 server.

public string Service {get;}

This property returns the service type: “udcs” for UniData or “uvcs” for UniVerse.

public int Timeout {get; set;}

This property gets or sets the length of time before the session times out during `ReadBlk()` operations. The remote procedure call (UniRPC) utility uses the timeout setting.

`int` is the timeout value in seconds. The default value is 300 seconds (5 minutes).

Note: If you enter a value that is too small, a running process may time out. If this occurs, an error code is returned and the connection to the server is dropped.

If this property fails, it throws a `UniSessionException`.

This property corresponds to the UniObjects **Timeout** property.

public int Transport {get; set;}

This property gets or sets the transport type to use when connecting to a server. Currently only TCP/IP connections are allowed.

`int` is the token number for the transport type, as follows:

Token number	Token	Description
0	UniObjectsTokens.NETWORK_DEFAULT	TCP/IP

Note: With TCP/IP connections, you must enter security information to connect to the server, for example, user name and password.

This property corresponds to the UniObjects **Transport** property.

public System.Text.Encoding UOEncoding {get; set;}

This property gets or sets the encoding object.

public string UserName {get;}

This property returns the user name to be used for the session connection operation.

It corresponds to the UniObjects **UserName** property.

UniSession – public instance methods

This section describes the public instance methods you can use with `UniSession` objects.

public string ByteArrayToUniCodeString (byte[ ] ByteArray)

This method converts the specified ByteArray to a UniCodeString per the encoding set in the `UOEncoding` property.

public Object Clone()

Creates a shallow copy of the `UniSession` object.

public UniSequentialFile CreateSequentialFile (string pFileName, string pRecordID, bool pCreateFlag)

Note: UniObjects for .NET cannot process UniData files sequentially. The following method applies only to UniVerse databases.

This method creates a file for sequential processing and returns a `UniSequentialFile` object.

`string pFileName` is the name of an existing type 1 or type 19 file.

`string pRecordID` is a record in the file. If the record does not exist and if `pCreateFlag` is true, this method creates a record.

`bool pCreateFlag` is a flag specifying that the record should or should not be created if it does not exist. If `pCreateFlag` is true, this method creates a record.

If the record cannot be opened because of an error on the server, `CreateSequentialFile` throws a `UniSessionException`, and the `UniSession` object's status is one of the following values:

Value	Description
0	No record ID was found.
1	The specified file is not type 1 or type 19.
2	The specified file was not found.

This example opens the TEST2 program file for sequential processing:

```
UniSequentialFile uSeq = uSession.CreateSequentialFile("BP",
"TEST2", false);
```

If this method fails, it throws a `UniSessionException`.

This method corresponds to the database `CREATE` command (if the create flag is set to `true`), the UniObjects **OpenSequential** method, and the BASIC OPENSEQ statement.

public void CreateTaskLock (int aLockNum)

This method locks one of the database's 64 task locks.

For more information about task locks, see Task locks.

`int aLockNum` is the number, 0 through 63, of the task lock to be set.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **SetTaskLock** method and the BASIC LOCK statement.

```
uSession.CreateTaskLock( 4 );
```

public UniCommand CreateUniCommand( )

This method creates and returns a `UniCommand` object for the session.

If this method fails, it throws a `UniSessionException`.

public UniDataSet CreateUniDataSet( )

This method creates and returns a `UniDataSet` object attached to the `UniSession` object.

If this method fails, it throws a `UniSessionException`.

public UniDictionary CreateUniDictionary (string pFileName)

This method opens an existing UniData or UniVerse dictionary file and returns a `UniDictionary` object, allowing access to the file.

`string pFileName` is the name of the UniData or UniVerse dictionary file to be opened.

This example opens the dictionary of the ORDERS file:

```
UniDictionary uDict = uSession.CreateUniDictionary("ORDERS");
```

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **OpenDictionary** method and the BASIC OPEN statement.

public UniDynArray CreateUniDynArray ( )

public UniDynArray CreateUniDynArray (string)

This method returns a new `UniDynArray` object either as a default string or as the string specified by the `string` object.

`string` is the string objects that represents the data you want to converted into a dynamic array.

The returned `UniDynArray` object inherits the system delimiters associated with this session.

If this method fails, it throws a `UniSessionException`.

public UniFile CreateUniFile (string pFileName)

This method opens an existing UniData or UniVerse database file and creates and returns a `UniFile` object, allowing access to the file.

`string pFileName` is the name of the database file to open.

This example opens the ORDERS file:

```
UniFile uFile = uSession.CreateUniFile ("ORDERS");
```

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **OpenFile** method and the BASIC OPEN statement.

public UniNLSLocale CreateUniNLSLocale( )

If NLS is enabled on the server machine, this method returns an active `UniNLSLocale` object, which can then be used to manipulate server-side NLS locale settings. Use the `NLSEnabled` property to determine if NLS is enabled.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **NLSLocale** property.

public UniNLSMap CreateUniNLSMap( )

If NLS is enabled on the server machine, this method returns an active `UniNLSMap` object, which can then be used to manipulate server-side NLS map settings. The `UniNLSMap` object represents the state of the server-side NLS map. Use the `NLSEnabled` property to determine if NLS is enabled.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **NLSMap** property.

public UniSelectList CreateUniSelectList (int aSelectListNumber)

This method creates and returns a `UniSelectList` object representing one of the 11 UniVerse select lists.

`int aSelectListNumber` is the number, 0 through 10, of the select list to use.

This example creates active select list 0:

```
UniSelectList uSelect = uSession.CreateUniSelectList(0);
```

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **SelectList** method.

public UniSubroutine CreateUniSubroutine (string aSubName, int aNumArgs)

This method creates and returns a `UniSubroutine` object.

`string aSubName` is the name of the subroutine to be executed on the server.

`int aNumArgs` is the number of arguments that the server subroutine uses.

This example calls the subroutine READ.CONFIG:

```
UniSubroutine uSub = uSession.CreateUniSubroutine("READ.CONFIG",
3);
```

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **Subroutine** method.

public UniTransaction CreateUniTransaction( )

This method creates and returns a `UniTransaction` object, which allows transaction control on the session and modification of the session's transactional behavior.

If this method fails, it throws a `UniSessionException`.

It corresponds to the UniObjects **Transaction** property.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public string Encrypt (string aString)

This method is used to encrypt a specified string into an internal format that is understood by the server.

`string aString` is the string to be encrypted.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public UniDynArray GetAtVariable (int aTokenVal)

This method returns the current value of the specified @variable.

`int aTokenVal` is the token number for the @variable whose value is to be retrieved, as follows:

Token number	Token	BASIC @Variables
1	UniObjectsTokens.AT_LOGNAME	@LOGNAME
2	UniObjectsTokens.AT_PATH	@PATH
3	UniObjectsTokens.AT_USERNO	@USERNO
4	UniObjectsTokens.AT_WHO	@WHO
5	UniObjectsTokens.AT_TRANSACTION	@TRANSACTION
6	UniObjectsTokens.AT_DATA_PENDING	@DATA.PENDING
7	UniObjectsTokens.AT_USER_RETURN_CODE	@USER.RETURN.CODE
8	UniObjectsTokens.AT_SYSTEM_RETURN_CODE	@SYSTEM.RETURN.CODE
9	UniObjectsTokens.AT_NULL_STR	@NULL.STR
10	UniObjectsTokens.AT_SCHEMA	@SCHEMA

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **GetAtVariable** method.

public byte[ ] GetDelimitedByteArrayRecordID (int[ ] pFieldNumber, int pDelimiter)

public byte[ ] GetDelimitedByteArrayRecordID (string[ ] pRecord ID, int pDelimiter)

This method creates and returns a delimited byte array of record IDs for the specified array of field numbers or string and delimiter.

public string GetDelimitedString (int[ ] pFieldNumber, int pDelimiter)

public string GetDelimitedString (string[ ] pRecord ID, int pDelimiter)

This method creates and returns a delimited string for the specified array of integers or string values and delimiter.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public byte GetMarkCharacter (int aMarkChar)

This method returns the byte of the specified system delimiter mark. Call this method, especially with an NLS-enabled server, to determine the proper values for each system delimiter.

`int aMarkChar` is the token number for the system delimiter, as follows.

Token number	Description
0	Item mark
1	Field mark
2	Value mark
3	Subvalue mark
4	Text mark
5	SQL null mark

If this method fails, it throws a `UniSessionException`.

Following is a code sample using UO.NET to test the GetMarkCharacter method:

```
uonetSession = UniObjects.OpenSession(txtHostname.Text, txtUID.Text, txtPWD.Text,
 txtDatabase.Text, txtProcess.Text)
uonetnlsmap = uonetSession.CreateUniNLSMap()
uonetnlsmap.SetName("MS1252+MARKS") '
0,1,2,3,4,5: 1 for FM mark tmp =
Convert.ToString(uonetSession.GetMarkCharacter(1))
```

This method corresponds to the UniObjects **FM**, **IM**, **SQLNULL**, **SVM**, **TM**, and **VM** properties.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public string Iconv (string aString, string aConvCode)

This method converts an input string to an internal storage format defined by the conversion code. After using Iconv, you can use the `Status` property to determine the status of this method.

`string aString` is the string to convert.

`string aConvCode` is any BASIC ICONV conversion code.

The `Iconv` ( ) method sets the `UniSession` object's status to one of the following values:

Value	Description
0	The conversion was successful.
1	The string supplied was invalid.
2	The conversion code supplied was invalid.
3	Conversion of possibly invalid data was successful.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **Iconv** method and the BASIC `ICONV` function.

```
try{
    UniString iDate = uSession.Iconv( "12 Oct 96", "D2/" );
} catch (UniSessionException e )
{ ... deal with error...
}
```

public string Oconv (string aString, string aConvCode)

This method converts an output string from internal storage format to an output storage format defined by a conversion code. After using Oconv, you can use the `Status` property to determine the status of this method.

`string aString` is the string to convert.

`string aConvCode` is any BASIC ICONV conversion code.

The `Oconv` ( ) method returns one of the following status values:

Value	Description
0	The conversion was successful.
1	The string supplied was invalid.
2	The conversion code supplied was invalid.
3	Successful conversion of possibly invalid data.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects `Oconv` method and the BASIC `OCONV` function.

```
try{
UniString oDate = uSession.Oconv( iDate, "D2/" );
} catch ( UniSessionException e )
{ ... deal with exception
}
```

public void ReleaseTaskLock (int pLockNum)

This method releases one of the 64 UniVerse task locks set previously through the `CreateTaskLock`() method.

For more information about task locks, see Task locks.

`int pLockNum` is the number, 0 through 63, of the task lock to release.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **ReleaseTaskLock** method.

```
uSession.ReleaseTaskLock( 4 );
```

public void SetAtVariable (int aTokenVal, UniDynArray aAtVariable)

This method sets the specified BASIC @variable to the value passed.

`int aTokenVal` is the token number for the @variable to be set. This method applies to the following @variable only:

Token number	Token	Description
7	UniObjectsTokens.AT_USER_RETURN_CODE	@USER.RETURN.CODE

`UniDynArray aAtVariable` is the UniDynArray value to which the @variable is to be set.

If this method fails, it throws a `UniSessionException`.

This method corresponds to the UniObjects **SetAtVariable** method.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

public byte[ ] UniCodeStringToByteArray (string pStringVal)

This method converts a UniCodeString to a byte array per the encoding set in the `UOEncoding` property.

`string pStringVal` is the string to be converted to a byte array.

If this method fails, it throws an `ArgumentNullException`.

UniSession – protected instance methods

This section lists the protected instance methods you can use with `UniSession` objects.

protected override void Dispose (bool disposing)

This method overrides the `Dispose`() method. It marks the session as disposed, but does not physically disconnect it from the database.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniSession object

```
UniSession us=null;
                try
{us =
UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");
UniCommand runCmd = uSession.CreateUniCommand( ) ;
runCmd.Command =  "RUN BP FOO" ;
runCmd.Execute();

}

                catch(Exception e)
                {
                    Console.WriteLine(e.Message +e.StackTrace);
                }
                finally
                {
                    if(us != null && us.IsActive)
                    {
                        UniObjects.CloseSession(us);
                    }
                }
```

UniFile class

The `UniFile` class defines and manages a data file on the server. You define the `UniFile` object through the `UniSession.CreateUniFile()` method.

For more information about creating and using a `UniFile` object, see Using files.

UniFile – public instance properties

This section describes the public instance properties you can use with `UniFile` objects.

public int EncryptionType {get; set;}

This property is inherited from `UniFile`. It gets or sets the type of encryption to use for all operations on UniFile and UniDictionary objects.

`int` is the token number for the encryption type, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_ENCRYPT	Do not encrypt data. This is the default value.
1	UniObjectsTokens.UV_ENCRYPT	Encrypt data using internal database encryption.

public string FileName {get;}

This property returns the name of the database file supplied by the `UniSession.CreateUniFile ( )` method.

It corresponds to the UniObjects **FileName** property.

public int FileStatus {get;}

This property gets the status code of the last method performed on this object. Refer to each method for a description of these status values.

This property corresponds to the UniObjects **Status** property.

public int FileType {get;}

This property returns the file type of the current file.

`int` is the file type. Valid file types are:

- 2 through 18 (static hashed files)
- 1 or 19 (nonhashed files)
- 25 (B-tree files)
- 30 (dynamic hashed files)

This property corresponds to the UniObjects **FileType** property.

public bool IsFileOpen {get;}

This property checks to see if a file is open. It returns true if file is open, or false if the file is closed.

For example:

```
UniSession us=null;
                try
                {
us =
UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");

UniFile fl = us.CreateUniFile("CUSTOMER");
If ( fl.IsFileOpen )
{
                    Console.WriteLine("Do Something");
}

}

                catch(Exception e)
                {
                    Console.WriteLine(e.Message +e.StackTrace);
                }
                finally
                {
                    if(us != null && us.IsActive)
                    {
                        UniObjects.CloseSession(us);
                    }
                }
```

This property corresponds to the UniObjects **IsOpen** method.

public UniDynArray Record {get; set;}

This property gets the contents of the record that was last read as a `UniDynArray`. It is updated whenever a `Read` ( ), `ReadField` ( ), or `ReadNamedField` ( ) method is called.

This property also sets the data portion of the record, primarily to be used for subsequent `Write` methods.

This property corresponds to the UniObjects **Record** property.

public string RecordID {get; set;}

This property gets the ID of the record that was last read. It is updated whenever a `Read` ( ), `ReadField` ( ), or `ReadNamedField` ( ) method is called.

This property also sets the record ID of the record to be read.

If this property fails, it throws a `UniFileException`.

It corresponds to the UniObjects **RecordId** property.

public string RecordString {get; set;}

This property gets the contents of the record that was last read as a string. It is updated every time a `Read`(), `ReadField`(), or `ReadNamedField`() is performed. This property also sets the data portion of the record, primarily to be used for subsequent `Write` methods.

public int UniFileBlockingStrategy {get; set}

This property gets or sets the UniFile blocking strategy, which is the action taken when a record or file lock blocks a database file operation.

The initial value is inherited from the `UniSession.BlockingStrategy` property. If you do not specify a value with the `UniFileBlockingStrategy` property, the value of the `UniSession.BlockingStrategy` property is used.

Use the `UniFileBlockingStrategy` property with the `UniFileLockStrategy` and `UniFileReleaseStrategy` properties.

int is the token number for the blocking strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.WAIT_ON_LOCKED	If the record is locked, wait until it is released (see Note).
2	UniObjectsTokens.RETURN_ON_LOCKED	Return a status value to indicate the state of the lock. This is the default value.

Note: Use the UniObjectsTokens.WAIT_ON_LOCKED value with caution. While the method is waiting for the lock to be released, your client window is effectively frozen and will not respond to mouse clicks.

If this property fails, it throws a `UniFileException`.

This property corresponds to the UniObjects **BlockingStrategy** property.

public int UniFileLockStrategy {get; set}

This property gets or sets the lock strategy, which controls the manner in which locks are set during `Read` operations on a file.

The initial value is inherited from the `UniSession.LockStrategy` property. If you do not specify a value with the `UniFileLockStrategy` property, the value of the `UniSession.LockStrategy` property is used.

`int` is the token number for the lock strategy, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_LOCKS	No locking. This is the default value.
1	UniObjectsTokens.EXCLUSIVE_READ	Sets an exclusive update lock (READU) for all file access.
2	UniObjectsTokens.SHARED_READ	Sets a shared read lock (READL) for all file access.

If this property fails, it throws a `UniFileException`.

Use this property with the `UniFileBlockingStrategy` and `UniFileReleaseStrategy` properties.

public int UniFileReleaseStrategy {get; set}

This property gets or sets the UniFile release strategy for releasing locks set by the `read()`, `readField()`, and `readNamedField()` methods and calls to the `LockRecord()` method.

The initial value is inherited from the `UniSession.ReleaseStrategy` property. If you do not specify a value with the `UniFileLockStrategy` property, the value of the `UniSession.ReleaseStrategy` property is used.

int is the token number for the release strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.WRITE_RELEASE	Releases the lock when the record is written. This is the default value.
2	UniObjectsTokens.READ_RELEASE	Releases the lock after the record is read.
4	UniObjectsTokens.EXPLICIT_RELEASE	Maintains locks as specified by the `UniFileLockStrategy` property. Locks can be released only with the `UnlockRecord()` method.
8	UniObjectsTokens.CHANGE_RELEASE	Releases the lock whenever a new value is set via the `RecordID()` property.

All the values are additive. If you specify UniObjectsTokens.EXPLICIT_RELEASE with UniObjectsTokens.WRITE_RELEASE and UniObjectsTokens.READ_RELEASE, it takes a lower priority. The initial release strategy value is 12, that is, release locks when the value of the record ID changes or when locks are released explicitly.

Use this property with the `UniFileBlockingStrategy` and `UniFileLockStrategy` properties.

If this property fails, it throws a `UniFileException`.

This property corresponds to the UniObjects **ReleaseStrategy** property.

UniFile – public instance methods

This section describes the public instance methods you can use with `UniFile` objects.

public void ClearFile ( )

This method clears a file, deleting all its records. If the file is locked by another session or user, the current blocking strategy (as returned by the `UniFileBlockingStrategy` property) determines the action to be taken.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ClearFile** method and the BASIC CLEARFILE statement.

public void Close ( )

This method closes a file and releases all file or record locks.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **CloseFile** method and the BASIC CLOSE statement.

public void DeleteRecord ( )

public void DeleteRecord (string aRecordIDObj)

public void DeleteRecord (UniDataSet aDataSet)

This method deletes a record.

`string aRecordIDObj` is the record ID of the record to be deleted.

`UniDataSet aDataSet` is a UniDataSet collection that identifies the record IDs to be deleted.

If you do not specify a record, the value set by the `RecordID` property is used.

This example deletes a record `rec` from the file `uFile`:

```
uFile.DeleteRecord(rec);
```

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **DeleteRecord** method and the BASIC DELETE and DELETEU statements.

See UniDataSet class for more details.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public UniDynArray GetAkInfo (string akNameObj)

This method returns information about the secondary key indexes in a `UniFile` object as a `UniDynArray` object. Value marks separate elements of the dynamic array.

`string akNameObj` is the field name of the secondary index whose information you want.

The meaning of the result depends on the type of index, as follows:

- For D-type indexes: field 1 contains D as the first character and field 2 contains the location number of the indexed field.
- For I-type indexes: field 1 contains I as the first character, field 2 contains the I-type expression, and the compiled I-type code occupies fields 19 onward.
- For both D-type and I-type indexes:
  - The second value of field 1 is 1 if the index needs to be rebuilt, or an empty string otherwise.
  - The third value of field 1 is 1 if the index is null-suppressed, or an empty string otherwise.
  - The fourth value of field 1 is 1 if automatic updates are disabled, or an empty string otherwise.
  - The sixth value of field 1 contains an S if the index is single-valued or an M if it is multivalued.

If `akNameObj` is an empty string, a list of available secondary key indexes on the file returns as a dynamic array of fields.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **GetAkInfo** method and the BASIC INDICES function.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public bool IsRecordLocked ( )

public bool IsRecordLocked (string aRecordIDObj)

This method indicates whether a user or session has locked a specified record.

`string aRecordIDObj` is the ID of the record to be checked. If `aRecordIDObj` is not specified, the value set by the `RecordID` property is used.

If this method fails, it throws a `UniFileException`.

public UniDynArray iType (string aRecordID, string aITypeID)

This method evaluates the specified I-descriptor and returns the evaluated string. It applies no conversions to the data.

`string aRecordID` is the record ID of the record supplied as data to the Itype facility.

`string aITypeID` is the record ID of the I-descriptor to be evaluated.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **IType** method and the BASIC ITYPE function.

public void LockFile ( )

This method locks the UniData or UniVerse file. It does not rely on any of the locking strategies such as those set by the `UniFileBlockingStrategy`, `UniFileLockStrategy`, or `UniFileReleaseStrategy` property. If another user or session has locked the file, `LockFile ( )` throws a `UniFileException`.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **LockFile** method and the BASIC FILELOCK statement.

public void LockRecord (int aLockFlag)

public void LockRecord (string aRecordID, int aLockFlag)

public void LockRecord (UniDataSet aDataSet, int aLockFlag)

This method locks a record; it sets the type of lock specified by `aLockFlag`. Use this method to override the current locking strategy.

`int aLockFlag` is the token number for the lock flag value, as follows:

Token number	Token	Description
1	UniObjectsTokens.EXCLUSIVE_UPDATE	Sets an exclusive update lock (READU).
2	UniObjectsTokens.SHARED_READ	Sets a shared read lock (READL).

`string aRecordID` specifies a record to be locked.

`UniDataSet aDataSet` specifies a dataset containing the records to be locked.

If you do not specify a record ID or dataset, the record is the one set previously by the `RecordID` property.

Using this method is equivalent to calling the `Read ( )`, `ReadField ( )`, or `ReadNamedField ( )` methods with the lock strategy set to the value of `aLockFlag`. If the value of `aLockFlag` is not valid, the method returns without performing any locking.

Note: You may need to explicitly unlock the record using the `UnlockRecord ( )` method, depending upon the release strategy value.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **LockRecord** method and the BASIC RECORDLOCKL and RECORDLOCKU statements.

public void Open ( )

This method opens a data file.

If `Open ( )` cannot open the file, it throws `UniFileException`.

public UniDynArray Read ( )

public UniDynArray Read (string aRecordID)

This method reads a database record and returns the data as a `UniDynArray` object.

`string aRecordID` is the ID of the record to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

The following example reads record 54637 in the ORDERS file:

```
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.Read("54637");
```

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **Read** method and the BASIC READ, READL, and READU statements.

public UniDynArray ReadField (int aFieldNumber)

public UniDynArray ReadField (string aRecordID, int aFieldNumber)

This method reads a field value from a database record.

`int aFieldNumber` is the number of the field to read. Specify field 0 (the record ID) to check if a record exists.

`string aRecordID` is the ID of the record whose field value you want to read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadField** method and the BASIC READV, READVL, and READVU statements.

```
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadField("2",3);
```

public UniDynArray ReadFields (int[ ] aFieldNumberSet)

public UniDynArray ReadFields (string aRecordID, int[ ] aFieldNumberSet)

This method is inherited from `UniFile`. It reads a specified array of fields from a UniData or UniVerse record.

`int[] aFieldNumberSet` is the array of field numbers to be read.

`string aRecordID` is the ID of the record whose field value is to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

```
int [] parr = {1,2,3};
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadFields("2",parr);
```

public UniDynArray ReadNamedField (string aFieldName)

public UniDynArray ReadNamedField (string aFieldName)

This method reads the value of a named field from a UniData or UniVerse record. It does this by extracting the field number from the dictionary associated with this file, and then performing a `ReadField` on that field.

`string aFieldName` is the name of the field to be read. The field must be defined by a D-descriptor or an I-descriptor in the file dictionary.

`string aRecordID` is the ID of the record containing the field to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

Note: This method needs to read the file dictionary in order to determine the location of the field. This can affect the performance of your application. If performance is an issue, use the `ReadField` ( ) method. For more information about using the `ReadNamedField` ( ) method, see Data conversion.

If `ReadNamedField` ( ) returns the error UVE_RNF (record not found), the missing record can be either the data record whose field value you want to read or the dictionary record defining the field.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadNamedField** method.

```
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadNamedField("2","FNAME");
```

public UniDynArry ReadNamedFields (string [ ] pFieldNames)

public UniDynArray ReadNamedFields (string pRecordID, string[ ] pFieldNames)

This method reads an array of fields identified by the named fields in `pFieldNames`. It does this by extracting the field numbers from the dictionary associated with this file, and then performing a `ReadField` on that field.

`string[] pFieldNames` is the name of the field to be read. The field must be defined by a D-descriptor or an I-descriptor in the file dictionary.

`string pRecordID` is the ID of the record containing the field to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

Note: This method needs to read the file dictionary in order to determine the location of the field. This can affect the performance of your application. If performance is an issue, use the `ReadField` ( ) method. For more information about using the `ReadNamedField` ( ) method, see Data conversion.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadNamedFields** method.

```
string [] parr = {“LNAME”,”FNAME”,”ADDRESS”};
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadNamedFields(“2”,parr);
```

public UniDataSet ReadRecords (string[ ] aRecordIDSet)

public UniDataSet ReadRecords (string[ ] aRecordIDSet), int[ ] aFieldNumberSet

public UniDataSet ReadRecords (string[ ] aRecordIDSet), string[ ] aFieldNameSet

This method reads a specified set of records from a UniData or UniVerse file.

`string[] aRecordIDSet` is a list of record IDs to be read from the file.

`int[] aFieldNameSet` or `string[] aFieldNameSet` is a set of fields to be read from the records.

> **Note:** This method only accepts D-type dictionary records to identify fields. If you need to used I-type dictionary records. use the IType method.

If this method fails, it throws a `UniFileException`.

```
string [] recID = {2,3,4,6,7};
string [] fieldNumbers = {1,2,3};

string [] fieldNames = {“LNAME”,”FNAME”,”ADDRESS”};

UniFile fl = us.CreateUniFile("CUSTOMER");

UniDataSet uds1 = fl.ReadRecords(recID);
UniDataSet uds2 = fl.ReadRecords(recID, fieldNumbers);
UniDataSet uds3 = fl.ReadRecords(recID, filedNames);
```

public UniDataSet ReadRecords2 (string[ ] aRecordIDSet)

public UniDataSet ReadRecords2 (string[ ] aRecordIDSet), int[ ] aFieldNumberSet

public UniDataSet ReadRecords2 (string[ ] aRecordIDSet), string[ ] aFieldNameSet

This method is used in cases where the default record separator (@RM) cannot be used because it will cause conflict with the data. However, when this method is used, performance might be slower than the `ReadRecords` method. This function is implemented as a resolution for specific customer situations. It should not be used except under Rocket Support guidance and may be withdrawn in future releases.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

public void UnlockFile ( )

This method removes all file locks from a database file. It corresponds to the UniObjects **UnlockFile** method and the BASIC FILEUNLOCK statement.

```
uFile.UnlockFile();
```

If this method fails, it throws a `UniFileException`.

public void UnlockRecord ( )

public void UnlockRecord (string aRecord ID)

public void UnlockRecord (string[ ] aRecordIDSet)

This method unlocks a record (or records).

`string aRecordID` is the ID of the record to be unlocked.

`string[] aRecordIDSet` is a set of record IDs to be unlocked.

If you do not specify `aRecordIDObj` or `aRecordIDSet`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **UnlockRecord** method and the BASIC RELEASE statement.

```
uFile UnlockRecord( "REC3" );
```

public void Write ( )

public void Write (string aRecordID, UniDynArray aRecordData)

public void Write (string aRecordID, string aRecordData)

This method writes data to a record.

`string aRecordID` is the ID of the record to write to. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` method.

`string aRecordData` is the value to write to the record. If you do not specify `aRecordData`, the value to write is the one set previously by the `Record` property.

After executing the `Write` ( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **Write** method and the BASIC WRITE and WRITEU statements.

```
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.Write(recID,arr);

string  recID2 = "4";
string str = "bbb";
fl.Write(recID,str);
```

public void WriteField (int aFieldNumber, UniDynArray aRecordData)

public void WriteField (int aFieldNumber, string aRecordData)

public void WriteField (string aRecordID, int aFieldNumber)

public void WriteField (string aRecordID, int aFieldNumber, UniDynArray aRecordData)

public void WriteField (string aRecordID, int aFieldNumber, string aRecordData)

This method writes data to a single field in a record.

`int aFieldNumber` is the number of the field to which data is to be written. If you do not specify `aFieldNumber`, this method writes to field 1.

`UniDynArray aRecordData` or `string aRecordData` is the field value to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

`string aRecordID` is the ID of the record to which the data is to be written. If you do not specify aRecordID, the record ID is the one set previously by the RecordID property.

After executing the `Write Field`() method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteField** method and the BASIC WRITEV and WRITEVU statements.

This example writes the string NewFieldValue into field 3 of the record REC3.

```
int fieldnumber = 5;
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteField(recID, fieldnumber ,arr);
```

public void WriteFields (int[ ] aFieldNumberSet)

public void WriteFields (string aRecordID, int[ ] aFieldNumberSet)

public void WriteFields (string aRecordID, int[ ] aFieldNumberSet, UniDynArray aRecordData)

This method writes data to an array of fields in a record.

`int[] aFieldNumberSet` is the array of fields to which data is to be written.

`string aRecordID` is the ID of the record to which the data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

`UniDynArray aRecordData` is the array of field values to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

After executing the `Write Fields`( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

```
int [] fieldnumbers = {5,6,7};
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteFields(recID, fieldnumbers ,arr);
```

public void WriteNamedField (string aFieldName, UniDynArray aRecordData)

public void WriteNamedField (string aFieldName, string aRecordData)

public void WriteNamedField (string aRecordID, string aFieldName, UniDynArray aRecordData)

This method writes data to a named field in a record, performing any input conversion defined in the file dictionary for the field.

> **Note:** `WriteNamedField`( ) does not convert distinct values in a multivalued field.

`string aFieldName` is the name of the field to which data is to be written, as defined in the file dictionary.

`UniDynArray aRecordData` or `string aRecordData` is the field value to be written to the record.

`string aRecordID` is the ID of the record to which data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

After executing the `WriteNamed Field`( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteNamedField** method.

```
string  fieldname = "LNAME ";
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteNamedField(recID, fieldname,arr);
```

public void WriteNamedFields (string[ ] aFieldNameSet)

public void WriteNamedFields (string aRecordID, string[ ] aFieldNameSet)

public void WriteNamedFields (string aRecordID, string[ ] aFieldNameSet, UniDynArray aRecordData)

This method writes data to a set of named fields in a record, performing any input conversion defined in the file dictionary for the field.

Note: `WriteNamedFields()` does not convert distinct values in a multivalued field.

`string[] aFieldNameSet` is a set of field names to which data is to be written, as defined in the file dictionary.

`string aRecordID` is the ID of the record to which data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

`UniDynArray aRecordData` is the array of field values to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

After executing the `WriteNamed Fields()` method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteNamedField** method.

```
string[]   fieldnames ={ "LNAME ","FNAME"};
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteNamedFields(recID, fieldnames,arr);
```

public void WriteRecords (UniDataSet aDataSet)

public void WriteRecords (int[ ] aFieldNumberSet, UniDataSet aDataSet)

public void WriteRecords (string[ ] aFieldNameSet, UniDataSet aDataSet)

This method writes data to records in a UniData or UniVerse file.

`UniDataSet aDataSet` specifies a dataset containing the records to which data is to be written.

`int[] aFieldNumberSet` is the array of fields to which data is to be written.

`string[] aFieldNameSet` is a list of record IDs to which data is to be written.

If this method fails, it throws a `UniFileException`.

```
us.RecordID = "2";
UniDataSet uSet = us1.CreateUniDataSet();
                    uSet2.Add("2","aaa");
uSet2.Insert("3","bbb");
fl.WriteRecords(uSet);
```

UniFile – protected instance methods

This section lists the protected instance methods you can use with `UniFile` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniDictionary class

The `UniDictionary` class controls access to UniData and UniVerse dictionary files. It is an extension of the `UniFile` class with properties and methods specific to dictionary files.

For more information about dictionary files and how to use them, see The database environment and Using a dictionary. For more information about the fields in a dictionary, see *Universe System Description*.

UniDictionary – public instance properties

This section describes the public instance properties you can use with `UniDictionary` objects.

public int EncryptionType {get; set;}

This property is inherited from `UniFile`. It gets or sets the type of encryption to use for all operations on UniFile and UniDictionary objects.

`int` is the token number for the encryption type, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_ENCRYPT	Do not encrypt data. This is the default value.
1	UniObjectsTokens.UV_ENCRYPT	Encrypt data using internal database encryption.

public string FileName {get;}

This property is inherited from `UniFile`. It returns the name of the database file supplied by the `UniSession.CreateUniFile()` method.

It corresponds to the UniObjects **FileName** property.

public int FileStatus {get;}

This property is inherited from `UniFile`. It gets the status code of the last method performed on this object. Refer to each method for a description of these status values.

This property corresponds to the UniObjects **Status** property.

public int FileType {get;}

This property is inherited from `UniFile`. It returns the file type of the current file.

`int` is the file type. Valid file types are:

- 2 through 18 (static hashed files)
- 1 or 19 (nonhashed files)
- 25 (B-tree files)
- 30 (dynamic hashed files)

This property corresponds to the UniObjects **FileType** property.

public bool IsFileOpen {get;}

This property is inherited from `UniFile`. It checks to see if a file is open. It returns true if file is open, or false if the file is closed.

This property corresponds to the UniObjects **IsOpen** method.

public UniDynArray Record {get; set;}

This property is inherited from `UniFile`. It gets the contents of the record that was last read as a `UniDynArray`. It is updated whenever a `Read()`, `ReadField()`, or `ReadNamedField()` method is called.

This property also sets the data portion of the record, primarily to be used for subsequent `Write` methods.

This property corresponds to the UniObjects **Record** property.

public string RecordID {get; set;}

This property is inherited from `UniFile`. It gets the ID of the record that was last read. It is updated whenever a `Read` ( ), `ReadField` ( ), or `ReadNamedField` ( ) method is called.

This property also sets the record ID of the record to be read.

If this property fails, it throws a `UniFileException`.

It corresponds to the UniObjects **RecordId** property.

public string RecordString {get; set;}

This property is inherited from `UniFile`. It gets the contents of the record that was last read as a string. It is updated every time a `Read`(), `ReadField`(), or `ReadNamedField`() is performed. This property also sets the data portion of the record, primarily to be used for subsequent `Write` methods.

public int UniFileBlockingStrategy {get; set}

This property is inherited from `UniFile`. It gets or sets the `UniFile` blocking strategy, which is the action taken when a record or file lock blocks a database file operation.

The initial value is inherited from the `UniSession.BlockingStrategy` property. If you do not specify a value with the `UniFileBlockingStrategy` property, the value of the `UniSession.BlockingStrategy` property is used.

Use the `UniFileBlockingStrategy` property with the `UniFileLockStrategy` and `UniFileReleaseStrategy` properties.

int is the token number for the blocking strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.WAIT_ON_LOCKED	If the record is locked, wait until it is released (see Note).
2	UniObjectsTokens.RETURN_ON_LOCKED	Return a status value to indicate the state of the lock. This is the default value.

Note: Use the UniObjectsTokens.WAIT_ON_LOCKED value with caution. While the method is waiting for the lock to be released, your client window is effectively frozen and will not respond to mouse clicks.

If this property fails, it throws a `UniFileException`.

This property corresponds to the UniObjects **BlockingStrategy** property.

public int UniFileLockStrategy {get; set}

This property is inherited from `UniFile`. It gets or sets the lock strategy, which controls the manner in which locks are set during Read operations on a file.

The initial value is inherited from the `UniSession.LockStrategy` property. If you do not specify a value with the `UniFileLockStrategy` property, the value of the `UniSession.LockStrategy` property is used.

`int` is the token number for the lock strategy, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_LOCKS	No locking. This is the default value.
1	UniObjectsTokens.EXCLUSIVE_READ	Sets an exclusive update lock (READU) for all file access.
2	UniObjectsTokens.SHARED_READ	Sets a shared read lock (READL) for all file access.

If this property fails, it throws a `UniFileException`.

Use this property with the `UniFileBlockingStrategy` and `UniFileReleaseStrategy` properties.

public int UniFileReleaseStrategy {get; set}

This property is inherited from `UniFile`. It gets or sets the `UniFile` release strategy for releasing locks set by the `read` ( ), `readField` ( ), and `readNamedField` ( ) methods and calls to the `LockRecord` ( ) method.

The initial value is inherited from the `UniSession.ReleaseStrategy` property. If you do not specify a value with the `UniFileLockStrategy` property, the value of the `UniSession.ReleaseStrategy` property is used.

`int` is the token number for the release strategy, as follows:

Token number	Token	Description
1	UniObjectsTokens.WRITE_RELEASE	Releases the lock when the record is written. This is the default value.
2	UniObjectsTokens.READ_RELEASE	Releases the lock after the record is read.
4	UniObjectsTokens.EXPLICIT_RELEASE	Maintains locks as specified by the `UniFileLockStrategy` property. Locks can be released only with the `UnlockRecord` ( ) method.
8	UniObjectsTokens.CHANGE_RELEASE	Releases the lock whenever a new value is set via the `RecordID` ( ) property.

All the values are additive. If you specify UniObjectsTokens.EXPLICIT_RELEASE with UniObjectsTokens.WRITE_RELEASE and UniObjectsTokens.READ_RELEASE, it takes a lower priority. The initial release strategy value is 12, that is, release locks when the value of the record ID changes or when locks are released explicitly.

Use this property with the `UniFileBlockingStrategy` and `UniFileLockStrategy` properties.

If this property fails, it throws a `UniFileException`.

This property corresponds to the UniObjects **ReleaseStrategy** property.

UniDictionary – public instance methods

This section describes the public instance methods you can use with `UniDictionary` objects.

public void ClearFile ( )

This method is inherited from `UniFile`. It clears a file, deleting all its records. If the file is locked by another session or user, the current blocking strategy (as returned by the `UniFileBlockingStrategy` property) determines the action to be taken.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ClearFile** method and the BASIC CLEARFILE statement.

public void Close ( )

This method is inherited from `UniFile`. It closes a file and releases all file or record locks.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **CloseFile** method and the BASIC CLOSE statement.

public void DeleteRecord ( )

public void DeleteRecord (string aRecordIDObj)

public void DeleteRecord (UniDataSet aDataSet)

This method is inherited from `UniFile`. It deletes a record.

`string aRecordIDObj` is the record ID of the record to be deleted.

`UniDataSet aDataSet` is a UniDataSet collection that identifies the record IDs to be deleted.

If you do not specify a record, the value set by the `RecordID` property is used.

This example deletes a record rec from the file `uFile`:

```
uFile.DeleteRecord(rec);
```

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **DeleteRecord** method and the BASIC DELETE and DELETEU statements.

See UniDataSet class for more details.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public UniDynArray GetAkInfo (string akNameObj)

This method is inherited from `UniFile`. It returns information about the secondary key indexes in a `UniFile` object as a `UniDynArray` object. Value marks separate elements of the dynamic array.

`string akNameObj` is the field name of the secondary index whose information you want.

The meaning of the result depends on the type of index, as follows:

- For D-type indexes: field 1 contains D as the first character and field 2 contains the location number of the indexed field.
- For I-type indexes: field 1 contains I as the first character, field 2 contains the I-type expression, and the compiled I-type code occupies fields 19 onward.
- For both D-type and I-type indexes:
  - The second value of field 1 is 1 if the index needs to be rebuilt, or an empty string otherwise.
  - The third value of field 1 is 1 if the index is null-suppressed, or an empty string otherwise.
  - The fourth value of field 1 is 1 if automatic updates are disabled, or an empty string otherwise.
  - The sixth value of field 1 contains an S if the index is single-valued or an M if it is multivalued.

If `akNameObj` is an empty string, a list of available secondary key indexes on the file returns as a dynamic array of fields.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **GetAkInfo** method and the BASIC INDICES function.

public UniDynArray GetAssoc ( )

This method returns the value in the ASSOC field (field 7) from the dictionary record set previously by the `RecordID` property.

This method corresponds to the UniObjects **ASSOC** property.

public UniDynArray GetConv ( )

public UniDynArray GetConv (string aRecordID)

This method returns the value in the CONV field (field 3) from a dictionary record.

`string aRecordID` is the ID of the dictionary record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **CONV** property.

public UniDynArray GetFormat ( )

public UniDynArray GetFormat (string aRecordID)

This method returns the value in the FORMAT field (field 5) from a dictionary record.

`string aRecordID` is the ID of the record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **FORMAT** property.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public UniDynArray GetLoc ( )

public UniDynArray GetLoc (string aRecordID)

This method returns the value in the LOC field (field 2) from a dictionary record.

`string aRecordID` is the ID of the dictionary record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **LOC** property.

public UniDynArray GetName ( )

public UniDynArray GetName (string aRecordID)

This method returns the value in the NAME field (field 4) from a dictionary record.

`string aRecordID` is the ID of the record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **NAME** property.

public UniDynArray GetSM ( )

public UniDynArray GetSM (string aRecordID)

This method returns the value in the SM field (field 6) from a dictionary record. The value in the SM field indicates whether the dictionary record is defined as single valued or multivalued.

`string aRecordID` is the ID of the record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **SM** property.

public UniDynArray GetSQLType ( )

public UniDynArray GetSQLType (string aRecordID)

This method returns the value in the SQLTYPE field from a dictionary record.

Note: This method applies only to UniVerse.

`string aRecordID` is the ID of the record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **SQLTYPE** property.

public UniDynArray GetSQLType ( )

new public UniDynArray GetType ( )

public UniDynArray GetType (string aRecordID)

This method is overloaded. It returns the value in the CODE field (field 1) from the dictionary record.

`string aRecordID` is the ID of the record to be evaluated. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

The first characters of the CODE field indicate the type of field the dictionary record is defining. Valid types are:

D	D-descriptor
I	I-descriptor
V	(UniData only) V-descriptor
PH	Phrase
X	(UniVerse only) X-descriptor

This method corresponds to the UniObjects **TYPE** property.

public bool IsRecordLocked ( )

public bool IsRecordLocked (string aRecordIDObj)

This method is inherited from `UniFile`. It indicates whether a user or session has locked a specified record.

`string aRecordIDObj` is the ID of the record to be checked. If `aRecordIDObj` is not specified, the value set by the `RecordID` property is used.

If this method fails, it throws a `UniFileException`.

public UniDynArray iType (string aRecordID, string aITypeID)

This method is inherited from `UniFile`. It evaluates the specified I-descriptor and returns the evaluated string. It applies no conversions to the data.

`string aRecordID` is the record ID of the record supplied as data to the Itype facility.

`string aITypeID` is the record ID of the I-descriptor to be evaluated.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **IType** method and the BASIC ITYPE function.

public void LockFile ( )

This method is inherited from `UniFile`. It locks the UniData or UniVerse file. It does not rely on any of the locking strategies such as those set by the `UniFileBlockingStrategy`, `UniFileLockStrategy`, or `UniFileReleaseStrategy` property. If another user or session has locked the file, `LockFile ( )` throws a `UniFileException`.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **LockFile** method and the BASIC FILELOCK statement.

public void LockRecord (int aLockFlag)

public void LockRecord (string aRecordID, int aLockFlag)

public void LockRecord (UniDataSet aDataSet, int aLockFlag)

This method locks a record; it sets the type of lock specified by `aLockFlag`. Use this method to override the current locking strategy.

`int aLockFlag` is the token number for the lock flag value, as follows:

Token number	Token	Description
1	UniObjectsTokens.EXCLUSIVE_UPDATE	Sets an exclusive update lock (READU).
2	UniObjectsTokens.SHARED_READ	Sets a shared read lock (READL).

`string aRecordID` specifies a record to be locked.

`UniDataSet aDataSet` specifies a dataset containing the records to be locked.

If you do not specify a record ID or dataset, the record is the one set previously by the `RecordID` property.

Using this method is equivalent to calling the `Read ( )`, `ReadField ( )`, or `ReadNamedField ( )` methods with the lock strategy set to the value of `aLockFlag`. If the value of `aLockFlag` is not valid, the method returns without performing any locking.

Note: You may need to explicitly unlock the record using the `UnlockRecord ( )` method, depending upon the release strategy value.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **LockRecord** method and the BASIC RECORDLOCKL and RECORDLOCKU statements.

public void Open ( )

This method is inherited from `UniFile`. It opens a data file.

If `Open ( )` cannot open the file, it throws `UniFileException`.

public UniDynArray Read ( )

public UniDynArray Read (string aRecordID)

This method reads a database record and returns the data as a `UniDynArray` object.

`string aRecordID` is the ID of the record to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

The following example reads record 54637 in the ORDERS file:

```
UniFile fl = us.CreateUniFile("CUSTOMER");
```

```
UniDynArray ar = fl.Read("54637");
```

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **Read** method and the BASIC READ, READL, and READU statements.

public UniDynArray ReadField (int aFieldNumber)

public UniDynArray ReadField (string aRecordID, int aFieldNumber)

This method is inherited from `UniFile`. It reads a field value from a database record.

`int aFieldNumber` is the number of the field to read. Specify field 0 (the record ID) to check if a record exists.

`string aRecordID` is the ID of the record whose field value you want to read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadField** method and the BASIC READV, READVL, and READVU statements.

```
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadField("2",3);
```

public UniDynArray ReadFields (int[ ] aFieldNumberSet)

public UniDynArray ReadFields (string aRecordID, int[ ] aFieldNumberSet)

This method is inherited from `UniFile`. It reads a specified array of fields from a UniData or UniVerse record.

`int[] aFieldNumberSet` is the array of field numbers to be read.

`string aRecordID` is the ID of the record whose field value is to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

```
int [] parr = {1,2,3};
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadFields("2",parr);
```

public UniDynArray ReadNamedField (string aFieldName)

public UniDynArray ReadNamedField (string aRecordID, string aFieldName)

This method is inherited from `UniFile`. It reads the value of a named field from a UniData or UniVerse record. It does this by extracting the field number from the dictionary associated with this file, and then performing a `ReadField` on that field.

`string aFieldName` is the name of the field to be read. The field must be defined by a D-descriptor or an I-descriptor in the file dictionary.

`string aRecordID` is the ID of the record containing the field to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

> **Note:** This method needs to read the file dictionary in order to determine the location of the field. This can affect the performance of your application. If performance is an issue, use `theReadField ( )` method. For more information about using the `ReadNamedField ( )` method, see Data conversion.

`If ReadNamedField ( )` returns the error UVE_RNF (record not found), the missing record can be either the data record whose field value you want to read or the dictionary record defining the field.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadNamedField** method.

```
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadNamedField("2","FNAME");
```

public UniDynArray ReadNamedFields (string[ ] pFieldNames)

public UniDynArray ReadNamedFields (string pRecordID, string[ ] pFieldNames)

This method is inherited from `UniFile`. It reads an array of fields identified by the named fields in `pFieldNames`. It does this by extracting the field numbers from the dictionary associated with this file, and then performing a `ReadField` on that field.

`string[] pFieldNames` is the name of the field to be read. The field must be defined by a D-descriptor or an I-descriptor in the file dictionary.

`string pRecordID` is the ID of the record containing the field to be read. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

> **Note:** This method needs to read the file dictionary in order to determine the location of the field. This can affect the performance of your application. If performance is an issue, use the `ReadField ( )` method. For more information about using the `ReadNamedField ( )` method, see Data conversion.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **ReadNamedFields** method.

```
string [] parr = {"LNAME","FNAME","ADDRESS"};
UniFile fl = us.CreateUniFile("CUSTOMER");

UniDynArray ar = fl.ReadNamedFields("2",parr);
```

public UniDataSet ReadRecords (string[ ] aRecordIDSet)

public UniDataSet ReadRecords2 (string[ ] aRecordIDSet), int[ ] aFieldNumberSet

public UniDataSet ReadRecords (string[ ] aRecordIDSet), string[ ] aFieldNameSet

This method reads a specified set of records from a UniData or UniVerse file.

`string[] aRecordIDSet` is a list of record IDs to be read from the file.

`int[] aFieldNameSet` or `string[] aFieldNameSet` is a set of fields to be read from the records.

> **Note:** This method only accepts D-type dictionary records to identify fields. If you need to used I-type dictionary records. use the IType method.

If this method fails, it throws a `UniFileException`.

```
string [] recID = {2,3,4,6,7};
string [] fieldNumbers = {1,2,3};

string [] fieldNames = {“LNAME”,”FNAME”,”ADDRESS”};

UniFile fl = us.CreateUniFile("CUSTOMER");

UniDataSet uds1 = fl.ReadRecords(recID);
UniDataSet uds2 = fl.ReadRecords(recID, fieldNumbers);
UniDataSet uds3 = fl.ReadRecords(recID, filedNames);
```

public void SetAssoc (UniDynArray aString)

public void SetAssoc (string aRecordID, UniDynArray aString)

This method sets the value of the ASSOC field (field 7) of a dictionary record.

`UniDynArray aString` is the value to be written to the ASSOC field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **ASSOC** property.

public void SetConv (UniDynArray aString)

public void SetConv (string aRecordID, UniDynArray aString)

This method sets the value of the CONV field (field 3) of a dictionary record.

`UniDynArray aString` is the value to write to the CONV field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **CONV** property.

public void SetFormat (UniDynArray aString)

public void SetFormat (string aRecordID, UniDynArray aString)

This method sets the value of the FORMAT field (field 5) of a dictionary record.

`UniDynArray aString` is the value to be written to the FORMAT field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **FORMAT** property.

public void SetLoc (UniDynArray aString)

public void SetLoc (string aRecordID, UniDynArray aString)

This method sets the value of the LOC field (field 2) of a dictionary record.

`UniDynArray aString` is the value to be written to the LOC field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **LOC** property.

public void SetName (UniDynArray aString)

public void SetName (string aRecordID, UniDynArray aString)

This method sets the value of the NAME field (field 4) of a dictionary record.

`UniDynArray aString` is the value to be written to the NAME field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **NAME** property.

public void SetSM (UniDynArray aString)

public void SetSM (string aRecordID, UniDynArray aString)

This method sets the value of the SM field (field 6) of a dictionary record. The value in the SM field indicates whether the dictionary record is defined as single valued or multivalued.

`UniDynArray aString` is the value to write to the SM field.

`string aRecordID` is the ID of the record you want. If you do not specify `aRecordID`, the record is specified by the `RecordID` property.

This method corresponds to the UniObjects **SM** property.

public void SetSQLType (UniDynArray aString)

public void SetSQLType (string aString)

public void SetSQLType (string aRecordID, UniDynArray aString)

public void SetSQLType (string aRecordID, string aString)

This method sets the value of the SQLTYPE field (field 8) of a dictionary record.

`UniDynArray aString` or `string aString` is the value to be written to the SQLTYPE field.

`string aRecordID` is the ID of the dictionary record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **SQLTYPE** property.

public void SetType (UniDynArray aString)

public void SetType (string aRecordID, UniDynArray aString)

This method sets the value of the CODE field (field 1) of the dictionary record.

`UniDynArray aString` is the value to be written to the CODE field. The first characters of the TYPE field indicate the type of field the dictionary record is defining. Valid types are:

D	D-descriptor
I	I-descriptor
V	(UniData only) V-descriptor
PH	Phrase
X	(UniVerse only) X-descriptor

`string aRecordID` is the ID of the record to be modified. If you do not specify `aRecordID`, the record is the one set previously by the `RecordID` property.

This method corresponds to the UniObjects **TYPE** property.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

public void UnlockFile ( )

This method is inherited from `UniFile`. It removes all file locks from a database file. It corresponds to the UniObjects **UnlockFile** method and the BASIC FILEUNLOCK statement.

```
uFile.UnlockFile();
```

If this method fails, it throws a `UniFileException`.

public void UnlockRecord ( )

public void UnlockRecord (string aRecord ID)

public void UnlockRecord (string[ ] aRecordIDSet)

This method is inherited from `UniFile`. It unlocks a record (or records).

`string aRecordID` is the ID of the record to be unlocked.

`string[] aRecordIDSet` is a set of record IDs to be unlocked.

If you do not specify `aRecordIDObj` or `aRecordIDSet`, the record ID is the one set previously by the `RecordID` property.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **UnlockRecord** method and the BASIC RELEASE statement.

```
uFile UnlockRecord( "REC3" );
```

public void Write ( )

public void Write (string aRecordID, UniDynArray aRecordData)

public void Write (string aRecordID, string aRecordData)

This method is inherited from `UniFile`. It writes data to a record.

`string aRecordID` is the ID of the record to write to. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` method.

`string aRecordData` is the value to write to the record. If you do not specify `aRecordData`, the value to write is the one set previously by the `Record` property.

After executing the `Write` ( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **Write** method and the BASIC WRITE and WRITEU statements.

```
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.Write(recID,arr);


string  recID2 = "4";
string str = "bbb";
fl.Write(recID,str);
```

public void WriteField (int aFieldNumber, UniDynArray aRecordData)

public void WriteField (int aFieldNumber, string aRecordData)

public void WriteFields (string aRecordID, int[] aFieldNumberSet)

public void WriteFields (string aRecordID, int[] aFieldNumberSet, UniDynArray aRecordData)

public void WriteField (string aRecordID, int aFieldNumber, string aRecordData)

This method is inherited from `UniFile`. It writes data to a single field in a record.

`int aFieldNumber` is the number of the field to which data is to be written. If you do not specify `aFieldNumber`, this method writes to field 1.

`UniDynArray aRecordData` or `string aRecordData` is the field value to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

`string aRecordID` is the ID of the record to which the data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

After executing the `Write Field`( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteField** method and the BASIC WRITEV and WRITEVU statements.

This example writes the string NewFieldValue into field 3 of the record REC3.

```
int fieldnumber = 5;
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteField(recID, fieldnumber ,arr);
```

public void WriteFields (int[ ] aFieldNumberSet)

public void WriteFields (string aRecordID, int[ ] aFieldNumberSet)

public void WriteFields (string aRecordID, int[ ] aFieldNumberSet, UniDynArray aRecordData)

This method is inherited from `UniFile`. It writes data to an array of fields in a record.

`int[] aFieldNumberSet` is the array of fields to which data is to be written.

`string aRecordID` is the ID of the record to which the data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

`UniDynArray aRecordData` is the array of field values to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

After executing the `Write Fields`( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

```
int [] fieldnumbers = {5,6,7};
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteFields(recID, fieldnumbers ,arr);
```

public void WriteNamedField (string aFieldName, UniDynArray aRecordData)

public void WriteNamedField (string aFieldName, string aRecordData)

public void WriteNamedField (string aRecordID, string aFieldName, UniDynArray aRecordData)

This method is inherited from `UniFile`. It writes data to a named field in a record, performing any input conversion defined in the file dictionary for the field.

Note: `WriteNamedField`( ) does not convert distinct values in a multivalued field.

`string aFieldName` is the name of the field to which data is to be written, as defined in the file dictionary.

`UniDynArray aRecordData` or `string aRecordData` is the field value to be written to the record.

`string aRecordID` is the ID of the record to which data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

After executing the `WriteNamed Field`( ) method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteNamedField** method.

```
string  fieldname = "LNAME ";
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
```

```
fl.WriteNamedField(recID, fieldname,arr);
```

public void WriteNamedFields (string[ ] aFieldNameSet)

public void WriteNamedFields (string aRecordID, string[ ] aFieldNameSet)

public void WriteNamedFields (string aRecordID, string[ ] aFieldNameSet, UniDynArray aRecordData)

This method is inherited from **UniFile**. It writes data to a set of named fields in a record, performing any input conversion defined in the file dictionary for the field.

Note: `WriteNamedFields()` does not convert distinct values in a multivalued field.

`string[] aFieldNameSet` is a set of field names to which data is to be written, as defined in the file dictionary.

`string aRecordID` is the ID of the record to which data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

`UniDynArray aRecordData` is the array of field values to be written to the record. If you do not specify `aRecordData`, the field value to be written is the one set previously by the `Record` property.

After executing the `WriteNamed Fields()` method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteNamedField** method.

```
string[] fieldnames ={ "LNAME ","FNAME"};
string recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteNamedFields(recID, fieldnames,arr);
```

public void WriteNamedField (string aFieldName, UniDynArray aRecordData)

public void WriteNamedField (string aFieldName, string aRecordData)

public void WriteNamedField (string aRecordID, string aFieldName, UniDynArray aRecordData)

This method is inherited from `UniFile`. It writes data to a named field in a record, performing any input conversion defined in the file dictionary for the field.

Note: `WriteNamedField()` does not convert distinct values in a multivalued field.

`string aFieldName` is the name of the field to which data is to be written, as defined in the file dictionary.

`UniDynArray aRecordData` or `string aRecordData` is the field value to be written to the record.

`string aRecordID` is the ID of the record to which data is to be written. If you do not specify `aRecordID`, the record ID is the one set previously by the `RecordID` property.

After executing the `WriteNamed Field()` method, call the `FileStatus` property to determine the state of record locks during the operation, as follows:

Value	Description
0	The record was locked before the operation.
–2	The record was not locked before the operation.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **WriteNamedField** method.

```
string  fieldname = "LNAME ";
string  recID = "2";
UniDynArray arr = new UniDynArray(us,"abc");
fl.WriteNamedField(recID, fieldname,arr);
```

public void WriteRecords (UniDataSet aDataSet)

public void WriteRecords (int[ ] aFieldNumberSet, UniDataSet aDataSet)

public void WriteRecords (string[ ] aFieldNameSet, UniDataSet aDataSet)

This method is inherited from `UniFile`. It writes data to records in a UniData or UniVerse file.

`UniDataSet aDataSet` specifies a dataset containing the records to which data is to be written.

`int[] aFieldNumberSet` is the array of fields to which data is to be written.

`string[] aFieldNameSet` is a list of record IDs to which data is to be written.

If this method fails, it throws a `UniFileException`.

```
us.RecordID = "2";
UniDataSet uSet = us1.CreateUniDataSet();
                    uSet2.Add("2","aaa");
uSet2.Insert("3","bbb");
fl.WriteRecords(uSet);
```

UniDictionary – protected instance methods

The section lists the protected instance methods you can use with UniDictionary objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniDictionary object

```
try
{
UniDictionary uFile = uSession.Create UniDictionary ("FOOBAR");
uFile.RecordID = "DTMTEST";
Console.WriteLine("Dictionaries entries for " +  uFile.RecordID);
Console.WriteLine ("DataValue = " + uFile.Record);
Console.WriteLine ("Assoc = " + uFile.GetAssoc());
Console.WriteLine ("Conversion = " + uFile.GetConv());
Console.WriteLine ("Format = " + uFile.GetFormat());
Console.WriteLine ("Loc = " + uFile.GetLoc());
Console.WriteLine ("Name = " + uFile.GetName());
Console.WriteLine ("SM = " + uFile.GetSM());
Console.WriteLine ("SQLTYPE = " + uFile.GetSQLType());
Console.WriteLine ("Type = " + uFile.GetType());
Console.WriteLine ("”);
Console.WriteLine ("Closing session ");
UniObjects.CloseSession(uSession);
productinfo/alldoc/UNh4
}
catch (Exception e)
{
Console.WriteLine(e.Message +e.StackTrace);
}
```

UniCommand class

The `UniCommand` class controls execution of database commands on the server. With it, users can run UniData or UniVerse commands or stored procedures on the server.

You can run only one command at a time during a session. For more information about using database commands, see Using database commands.

UniCommand – public instance properties

This section describes the public instance properties you can use with `UniCommand` objects.

public int Command ServerSessionPid {get}

You can get the process ID of the server process using this command.

Example:

```
UniCommand runCmd = uSession.command();

System.out.println(runCmd.getServerSessionPid());
```

public string Command {get; set;}

This property gets or sets the command string to be executed on the server. It corresponds to the UniObjects **Text** property.

This example sets up a database command for execution:

```
uvc.Command = "LIST VOC SAMPLE 10";
```

public int Command AtSelected {get;}

This property gets the value of the @SELECTED variable from the server when the command has completed successfully. It corresponds to the UniObjects **AtSelected** property.

public int Command BlockSize {get; set;}

This property gets or sets the block size, in bytes, of the buffer used to hold the contents of the `Response` property in server communications. The initial value is 0, which means that there is no limit to the size of the buffer, and all data is to be returned.

If you expect a command to generate large quantities of data, you can set the block size to a manageable value and read the output in blocks. You read successive blocks with the `NextBlock()` method. In this case, the `CommandStatus` property returns UVS_MORE when the buffer is full, and when you call the `Response` property, the next block of command output is read from the server.

Note: In a client/server application, running server commands that produce large quantities of output can decrease performance and increase network traffic. For more information on this topic, see Client/server design considerations.

This property corresponds to the UniObjects **BlockSize** property.

public int CommandStatus {get;}

This property gets the status of the command object execution. The status is one of the following:

Value	Token	Description
0	UniObjectsTokens.UVS_COMPLETE	The command finished execution or was cancelled. A new command can be executed.
1	UniObjectsTokens.UVS_REPLY	The server is waiting for input data. The reply can be sent using the `Reply()` method.

Value	Token	Description
2	UniObjectsTokens.UVS_MORE	More data is waiting to be retrieved. This occurs only if the block size is set to a non-zero value in the `CommandBlockSize` property.

If you use the `CommandBlockSize` property to set the block size to a value other than 0, the Response property returns a data segment equivalent to the size that is set. If the command results are more than can fit in one block, call the `NextBlock ( )` method until the `CommandStatus` property returns UVS_COMPLETE.

This property corresponds to the UniObjects **CommandStatus** property.

public int EncryptionType {get; set;}

This property gets or sets the default encryption type to be used for client-server communications in all `UniCommand` object operations, as follows:

Value	Token	Description
0	UniObjectsTokens.NO_ENCRYPT	Do not encrypt data. This is the default value.
1	UniObjectsTokens.UV_ENCRYPT	Encrypt data using internal database encryption.

If an encryption type is set, all data transferred between client and server for `UniCommand` objects is encrypted.

This property overrides the `UniSession` default `EncryptionType`.

public string Response {get;}

This property gets the output from the `Execute( )` and `Reply ( )` methods. This is the output generated by the command on the server.

This property corresponds to the UniObjects **Response** property.

public int SystemReturnCode {get;}

This property gets the value of the @SYSTEM.RETURN.CODE returned by the command on the server. It corresponds to the UniObjects **SystemReturnCode** property.

UniCommand – public instance methods

This section describes the public instance methods you can use with `UniCommand` objects.

public void Cancel ( )

This method cancels all outstanding output from the executing command. You can call this method only when the command status returned by the `CommandStatus` property is either UVS_REPLY or UVS_MORE. If the `Cancel ( )` method is successful, the command status is reset to UVS_COMPLETE, allowing another command to be executed.

If this method fails, it throws a `UniCommandException`.

This method corresponds to the UniObjects **Cancel** method.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public void Execute ( )

This method executes the command set up by the `Command` property.

Use the Response property to get the results from executing the command. If an error occurs, `Execute`( ) throws a `UniCommandException` and the `Response` property returns the error message produced by the executed command.

The `CommandStatus` property gets the current status of the command, that is, whether it has completed or is waiting for further input.

This example executes the command LIST VOC SAMPLE 10 on the server:

```
UniCommand runCmd = uSession.CreateUniCommand();
runCmd.Command = "LIST VOC SAMPLE 10";
runCmd.Execute();
```

If this method fails, it throws a `UniCommandException`.

This method corresponds to the UniObjects **Exec** method and the BASIC EXECUTE statement.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void NextBlock ( )

This method gets the next block of data from the command response if the command response exceeds the block size established by the `CommandBlockSize` property. After each execution of `NextBlock`, you can call the `Response` property to get the new block of data, and then call the `CommandStatus` property to determine the status of the command's execution.

If this method fails, it throws a `UniCommandException`.

This method corresponds to the UniObjects **NextBlock** method.

public void Reply (string aReplyString)

This method replies to a command execution that is currently in the UVS_REPLY state. Many commands require a user response. Use the `Reply ( )` method to issue the correct response to a command. Call this method whenever the `CommandStatus` property returns UVS_REPLY.

`string aReplyString` is the string to send to the server as a response.

If this method fails, it throws a `UniCommandException`.

This method corresponds to the UniObjects **Reply** method.

```
UniCommand runCmd = uSession.CreateUniCommand( ) ;
runCmd.Command =  "RUN BP FOO" ;
runCmd.Execute();
if ( runCmd.CommandStatus == UniObjectsTokens.UVS_REPLY )
{
/* Command may need to respond to a 'Press y to continue' */
runCmd.Reply( "Y" );
}
```

Note: The UO.NET driver was updated in the September 2015 client to support the UO.NET Reply method. Calling the Reply method will clear the Command objects Response property buffer.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniCommand – protected instance methods

This section describes the protected instance methods you can use with `UniCommand` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniCommand object

```
UniSession us=null;
                try
                {
us =
UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");
```

```
UniCommand runCmd = us.CreateUniCommand( ) ;
runCmd.Command =  "LIST VOC SAMPLE 10" ;
runCmd.Execute();
string reply = runCmd.Response;
Console.WriteLine(reply);
}
                catch(Exception e)
                {
                    Console.WriteLine(e.Message +e.StackTrace);
                }
                finally
                {
                    if(us != null && us.IsActive)
                    {
                        UniObjects.CloseSession(us);
                    }
                }
```

UniDataSet class

`UniDataSet` is a collection class. It provides a collection interface for sets of `UniRecord` objects, which can then be used to perform bulk or batch operations with one network operation.

The `UniDataSet` class has implemented the `System.Collections.IEnumerator` interface. The `foreach` statement offers a convenient way to iterate over the elements of `UniDataSet`.

UniDataSet – public instance constructors

This section describes the public instance constructors for the `UniDataSet` class.

public UniDataSet (UniSession pSession)

public UniDataSet (UniSession pSession, string[ ] RecId, byte[ ] RecData, byte[ ] StatusData, byte[ ] RetValData)

This initializes a new instance of the `UniDataSet` class.

`UniSession pSession` is a `UniSession` object.

`string[] RecId` is an array of record IDs.

`byte[] RecData` is a byte array of record data.

`byte[] StatusData` is a byte array of record status.

`byte[] RetValData` is a byte array of record return values.

If the second form of the constructor fails, it throws a `UniDataSetException`.

UniDataSet – public instance properties

This section describes the public instance properties you can use with `UniDataSet` objects.

public bool AfterLast {get;}

This property returns a Boolean value indicating whether the cursor is positioned past the last row in the data set. Use this method to determine when the list is exhausted.

public bool BeforeFirst {get;}

This property returns a Boolean value indicating whether the cursor is positioned before the first row in the data set.

public int CurrentRow {get; set;}

This property gets or sets the current index position within the UniDataSet collection object.

public byte[ ] DelimitedByteArrayRecord {get;}

This property returns a byte array object that represents all records, delimited by record marks.

public byte[ ] DelimitedByteArrayRecordID {get;}

This property returns a byte array object that represents all record IDs, delimited by record marks.

public string DelimitedRecord {get;}

This property returns a string object that represents all records, delimited by record marks.

public bool First {get;}

This property returns a Boolean value indicating whether the cursor is positioned at the first row in the data set.

public UniRecord this [int nIndex] {get;}

public UniRecord this [string RecID] {get; set}

This property is the indexer for the `UniDataSet` class. It gets or sets the value associated with the specified index key or record ID key.

`int nIndex` is the index key.

`string RecID` is the record ID key.

If this property fails, it throws a `UniDataSetException`.

public bool Last {get;}

This property returns a Boolean value indicating whether the cursor is positioned at the last row in the data set.

public int RowCount {get;}

This property gets the number of UniRecord objects contained in the UniDataSet collection object.

UniDataSet – public instance methods

This section describes the public instance methods you can use with `UniDataSet` objects.

public bool Absolute (int rowNum)

This method specifies the absolute position in the `UniDataSet` to which the cursor should point. It returns a Boolean value indicating whether the operation was successful.

`rowNum` is an integer specifying the absolute position.

public void Add (string pUniRecID)

public void Add (string pUniRecID, UniDynArray pUniRecord)

public void Add (string pUniRecID, UniRecord pUniRec)

public void Add (string pUniRecID, string pRecord)

This method adds a `UniRecord` object to the end of the `UniDataSet` collection object.

`string pUniRecID` is the record ID of the row to be added.

`UniDynArray pUniRecord` is the UniDynArray to be converted to a `UniRecord` object.

`UniRecord pUniRec` is the `UniRecord` object to be added.

string `pRecord` is the string to be converted to the `UniRecord` object.

If this method fails, it throws a `UniDataSetException`.

public void Clear( )

This method removes all elements from the `UniDataSet` collection object.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public IEnumerator GetEnumerator( )

This method returns an enumerator that can iterate through the `UniDataSet`.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public UniRecord GetRecord (int nIndex)

public UniRecord GetRecord (string pUniRecID)

This method gets the value associated with the specified index position or the specified record ID.

`int nIndex` is the index position whose value is to be retrieved.

`string pUniRecID` is the ID of the record whose value is to be retrieved.

If this method fails, it throws a `UniDataSetException`.

public int GetRecordStatus (int nIndex)

public int GetRecordStatus (string pUniRecID)

This method gets the `UniRecord` status associated with the specified index key or the specified record ID.

`int nIndex` is the index position for the associated `UniRecord` object whose status is to be determined.

`string pUniRecID` is the record ID for the `UniRecord` object whose status is to be retrieved.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void Insert (int pIndexLoc, string pUniRecID, UniDynArray pUniRecord)

public void Insert (int pIndexLoc, string pUniRecID, UniRecord pRecord)

public void Insert (int pIndexLoc, string pUniRecID, string pRecord)

public void Insert (string pUniRecID)

public void Insert (string pUniRecID, UniDynArray pRecord)

public void Insert (string pUniRecID, UniDynArray pRecord)

public void Insert (string pUniRecID, string pRecord)

This method inserts a new row into the data set at the specified cursor position. It returns a Boolean value indicating whether the operation was successful.

`int pIndexLoc` is the location at which the row is to be inserted in the dataset.

`string pUniRecID` is the record ID of the row to be inserted.

`UniDynArray pUniRecord` is the UniDynArray to be converted to a `UniRecord` object before being inserted.

`UniRecord pRecord` or `string pRecord` is the UniRecord or string to be converted to a `UniRecord` object before being inserted.

If this method fails, it throws a `UniDataSetException`.

public bool Relative (int numRows)

This method positions the data set cursor to a position `numRows` away from the current position. For example, if the cursor is already set to the third row and `UniDataSet.relative(5)` is referenced, the cursor is set to the eighth position in the data set. If the operation succeeds, this method returns true. If the operation attempts to move the cursor past the end or before the beginning of the data set, it returns false and sets the cursor to the last row or first row, respectively.

`numRows` is an integer representing the number of rows to move the cursor.

public void Remove (string pUniRecID)

This method removes the element with the specified record ID from the UniDataSet collection object.

`string pUniRecID` is the record ID of the element to be removed from the dataset.

public override string ToString ( )

This method returns a string that represents the `UniDataSet` collection object. This method overrides the `ToString` method of the `Object` class.

UniDataSet – protected instance methods

This section describes the protected instance methods you can use with `UniDataSet` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniDynArray class

The `UniDynArray` object lets you manipulate fields, values, and subvalues in a dynamic array such as a database record or a select list. `UniDynArray` objects are used in:

- Data in records of the `UniFile` and `UniDictionary` objects
- The `readList` method of the `UniSelectList` object

It handles database strings that contain field marks, value marks, and subvalue marks.

The `UniDynArray` class converts an input string into a series of subobjects, each of which is inserted into a .NET ARRAYLIST object. Because of this, the dynamic array needs to be parsed only once, and ARRAYLIST operations can easily manipulate the `UniDynArray` object.

For more information about the `UniDynArray` object, see Fields, values, and subvalues.

UniDynArray – public instance constructors

This section describes the public instance constructors for the `UniDynArray` class.

public UniDynArray (UniSession aSession)

This is the default constructor for the class. It constructs a dynamic array with no characters in it.

`aSession` is a `UniSession` object. If a `UniSession` object instantiates it, the `UniDynArray` object inherits the system delimiters defined for that session; otherwise, it uses the standard default system delimiters.

public UniDynArray (UniSession aSession, byte[ ] pData)

This syntax constructs a dynamic array containing the value of the byte array.

`aSession` is a `UniSession` object. If a `UniSession` object instantiates it, the `UniDynArray` object inherits the system delimiters defined for that session; otherwise, it uses the standard default system delimiters.

`byte[] pData` is the byte array data to be converted to a dynamic array.

If this constructor fails, it throws a `UniDynArrayException`.

public UniDynArray (UniSession aSession, string pString)

This syntax constructs a dynamic array containing the value of `pString`.

`aSession` is a `UniSession` object. If a `UniSession` object instantiates it, the `UniDynArray` object inherits the system delimiters defined for that session; otherwise, it uses the standard default system delimiters.

`string pString` is the data to be converted to a dynamic array.

UniDynArray – public instance properties

This section describes the public instance properties you can use with `UniDynArray` objects.

public string StringValue [get;}

Gets the value of a UniDynArray as a string object.

UniDynArray – public instance methods

This section describes the public instance methods you can use with `UniDynArray` objects.

public int Count ( )

public int Count (int aField)

public int Count (int aField, int aValue)

public int Count (int aField, int aValue, int aSubValue)

This method counts one of the following:

- The number of field marks in the `UniDynArray` object
- The number of value marks in a field of the `UniDynArray` object
- The number of subvalue marks in a value of the `UniDynArray` object
- The number of text marks in a subvalue of the `UniDynArray` object

`int a Field` is the field whose value marks, subvalue marks, or text marks are to be counted.

`int aValue` is the value whose subvalue marks or text marks are to be counted.

`int aSubValue` is the subvalue whose text marks are to be counted.

This method corresponds to the UniObjects **Count** method and the BASIC `COUNT` function.

public int Dcount ( )

public int Dcount (int aField)

public int Dcount (int aField, int aValue)

public int Dcount (int aField, int aValue, int aSubValue)

This method counts one of the following:

- The number of fields in the `array`, equivalent to `Count` ( ) +1
- The number of values at a specified field position in the `array`, equivalent to `Count` (`aField`) +1
- The number of subvalues in a specified field position/value position in the `array`, equivalent to `Count` (`aFieldValue, aValue`) +1
- The number text values in a specified field position/value position/subvalue position of the array, equivalent to `Count` ( ) +1

`int aField` is the field whose values, subvalues, or text values are to be counted.

`int aValue` is the value whose subvalues or text values are to be counted.

`int aSubValue` is the subvalue whose text values are to be counted.

It corresponds to the UniObjects **Count** method and the BASIC `DCOUNT` function.

public void Delete (int aField)

public void Delete (int aField, int aValue)

public void Delete (int aField, int aValue, int aSubValue)

This method deletes the specified field, value, or subvalue from a dynamic array.

`int aField` is the number of the field to be deleted, or the number of the field containing the value or subvalue to be deleted.

`int aValue` is the number of the value to be deleted, or the number of the value containing the subvalue to be deleted.

`int aSubValue` is the number of the subvalue to be deleted.

This method corresponds to the UniObjects **Del** method and the BASIC `DELETE` function.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public UniDynArray Extract (int aField)

public UniDynArray Extract (int aField, int aValue)

public UniDynArray Extract (int aField, int aValue, int aSubValue)

This method extracts one of the following:

- A field in a specified position of the `UniDynArray` object
- The value in a specified position of a field in the `UniDynArray` object
- The subvalue in a specified position of a value in a field in the `UniDynArray` object

`int aField` is the number of the field to be extracted, or the number of the field containing the value or subvalue to be extracted.

`int aValue` is the number of the value to be extracted, or the number of the value containing the subvalue to be extracted.

`int aSubValue` is the number of the subvalue to be extracted.

This method corresponds to the BASIC `EXTRACT` function.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void Insert (int aField, string aString)

public void Insert (int aField, int aValue, string aString)

public void Insert (int aField, int aValue, int aSubValue, string aString)

This method inserts a string object into a dynamic array at a specified field position/value position, moving subsequent fields, values, or subvalues down.

`int aField` is the number of the field into which data is to be inserted, or the number of the field into which a value or subvalue is to be inserted.

`string aString` is the string representing the data to be inserted.

`int aValue` is the number of the value to insert, or the number of the value into which the subvalue is to be inserted.

`int aSubValue` is the number of the subvalue to be inserted.

This method corresponds to the UniObjects **Ins** method and the BASIC `INSERT` function.

public int Length (int aField)

public int Length (int aField, int aValue)

public int Length (int aField, int aValue, int aSubValue)

This method gets the length of a specified field, value, or subvalue in a `UniDynArray` object.

`int aField` is the number of the field whose length is to be retrieved, or the number of the field containing the value or subvalue whose length is to be retrieved.

`int aValue` is the number of the value whose length you want, or the number of the value containing the subvalue whose length is to be retrieved.

`int aSubValue` is the number of the subvalue whose length is to be retrieved.

This method corresponds to the UniObjects **Length** method.

public void PrintByteArray( )

This method prints each byte of the byte array to the console.

public UniDynArray Remove (int aField)

public UniDynArray Remove (int aField, int aValue)

public UniDynArray Remove (int aField, int aValue, int aSubValue)

This method deletes a field, value, or subvalue from the `UniDynArray` object, returning the field, value, or subvalue as a new `UniDynArray` object.

`int aField` is the number of the field to be removed, or the number of the field containing the value or subvalue to be removed.

`int aValue` is the number of the value to be removed, or the number of the value containing the subvalue to be removed.

`int aSubValue` is the number of the subvalue to be removed.

public void Replace (int aField, string aString)

public void Replace (int aField, int aValue, string aString)

public void Replace (int aField, int aValue, int aSubValue, string aString)

This method replaces a field, value, or subvalue with a new field, value, or subvalue.

`int aField` is the number of the field whose value is to be replaced, or the number of the field containing the value or subvalue is to be replaced.

`int aValue` is the number of the value is to be replaced, or the number of the value containing the subvalue is to be replaced.

`int aSubValue` is the number of the subvalue is to be replaced.

`string aString` is the replacement string value.

This method corresponds to the UniObjects **Replace** method and the BASIC `REPLACE` function.

public byte[ ] ToByteArray( )

This method converts the specified `UniDynArray` object into a byte array.

public override string ToString ( )

This method converts the specified `UniDynArray` object into a base string.

UniDynArray – protected instance methods

This section lists the protected instance methods you can use with `UniDynArray` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniDynArray object

```
                try
                                        {
UniSession us =
UniObjects.OpenSession("localhost","ZZZ","XXX","HS.SALES","uvcs");
                    //creating UniDynArray
                    char bFM = Convert.ToChar(254);
                    char bVM = Convert.ToChar(253);
                    char bSVM = Convert.ToChar(252);
UniDynArray lDynArray =  new UniDynArray(us1,"ab" + bFM + "cd" + bVM + "ef"
+ bVM
                    + "gh" + bVM + "ij" + bFM + "kl" + bSVM + "mn" + bSVM + "no" +
                        bVM + "p" + bVM + "qr" + bFM + "s" + bFM + "t" + bFM + "");
                    // run  Count()
                    int myVal = lDynArray.Count();

                    // run Dcount()
                    int myVal2 = lDynArray.Dcount();
                                        // run Extract
                    UniDynArray real = lDynArray.Extract(1,1,0);
                    // run Replace
                    lDynArray.Replace(2, 0, 0, "*");
                    //run delete
                    lDynArray.Delete(1, 0, 0);
                    // run insert
                    lDynArray.Insert(0, 0, 0, "2500");
                    }
                    Catch (Exception ex)
                    {
                    //some error, display it
                    Console.WriteLine(ex.Message);
                    }
                    finally
                    {
                    // no error
                    if(us1 != null)
                    {
                    UniObjects.CloseSession(us1);
                    us1= null;
                    }
                    }
                                    }
```

UniNLSLocale class (UniVerse only)

The `UniNLSLocale` object applies only to UniVerse systems.

On UniVerse systems, the `UniNLSLocale` object defines and manages the National Language Support conventions in use. The five conventions are Time, Numeric, Monetary, Ctype, and Collate. The `UniNLSLocale` object allows these five names to be supplied as a single `UniDynArray` object, with five fields containing the relevant locale name. Locale names are derived from the client system and a defaultable locale identifier.

The `UniNLSLocale` object is available from the `UniSession` object via the `UniSession.CreateNLSLocale` ( ) method. If NLS is disabled on the server, the `UniNLSLocale` object is not available, and CreateNLSLocale ( ) throws an exception.

UniNLSLocale – public instance properties

This section describes the public instance properties you can use with `UniNLSLocale` objects.

public UniDynArray ClientNames {get;}

This method returns a `UniDynArray` of the locale names requested by the client. This is the locale specification as the client sees it.

This method corresponds to the UniObjects **ClientName** method.

public UniDynArray ServerNames {get;}

This property returns a UniDynArray of locale names as reported by the server. These can differ from the names returned by the `ClientNames` property because of a difference between client and server naming styles.

This property corresponds to the UniObjects **ServerName** method.

UniNLSLocale – public instance methods

This section describes the public instance methods you can use with `UniNLSLocale` objects.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void SetLocaleName (UniDynArray aName)

public void SetLocaleName (UniDynArray aName, int anIndex)

public void SetLocaleName (string aName)

public void SetLocaleName (string aName, int anIndex)

This method sets the specified locale.

If `aName` is of type `UniDynArray`, each category is set to the corresponding `UniDynArray` value. If the `UniDynArray` contains only one element and anIndex is specified, only that locale setting is changed. If anIndex is not specified, all locale categories are set to the value defined by `aName`.

If `aName` is of type `string`, each category is set to the corresponding string value. If the string contains only one element and `anIndex` is specified, only that locale setting is changed. If `anIndex` is not specified, all locale categories are set to the value defined by `aName`.

`UniDynArray aName` or `string aName` is a UniDynArray or string representing the new locale settings.

`int anIndex` is an integer representing the category for which the locale is to be set.

If this method fails, it throws a `UniTransactionException`.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniNLSLocale – protected instance methods

This section describes the protected instance methods you can use with `UniNLSLocale` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniNLSMap class (UniVerse only)

The `UniNLSMap` class applies only to UniVerse systems.

`UniNLSMap` controls NLS map settings. The UniVerse server uses NLS maps to determine which map to use for a client's string data.

The `UniNLSMap` object is available from the `UniSession` object. The `UniNLSMap` object is available only if NLS is enabled on the server uniConnection.

UniNLSMap – public instance properties

This section describes the public instance properties you can use with `UniNLSMap` objects.

public string ClientMapName

This property returns the name of the map requested by the client. On the server it is mapped through the NLS.CLIENT.MAPS file to the name reported by the `ServerMapName` property.

This method corresponds to the UniObjects **ClientName** property.

public string ServerMapName {get;}

This property returns the name of the map as reported by the server. This is the name that is loaded into the server shared memory segment. This value may be different from the name requested via the `GetClientMapName` method. because of client-server NLS map name mapping.

If this property fails, it throws a `UniNLSMapException`.

This property corresponds to the UniObjects **ServerName** property.

public byte UniMarks {get;}

This property gets marks from the server.

UniNLSMap – public instance methods

This section describes the public instance methods you can use with `UniNLSMap` objects.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void SetName (string pName)

This method sets the map to use on the server.

`string pName` is the name of the requested map.

When the name has been changed successfully, the `ServerMapName` property and `GetClientMapName`() method return the corresponding value.

If this method fails, it throws a `UniNLSMapException`.

This method corresponds to the UniObjects **SetName()** method.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniNLSMap – protected instance methods

This section lists the protected instance methods you can use with `UniNLSMap` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniRecord class

The `UniRecord` object controls database record interaction. It contains the `UniDynArray` object and RecordID.

UniRecord – public instance constructors

This section describes the public instance constructor for the `UniRecord` class.

public UniRecord ( )

This constructs an instance of the `UniRecord` class with no data in it.

UniRecord – public instance properties

This section describes the public instance properties you can use with `UniRecord` objects.

public UniDynArray Record {get; set;}

This property gets or sets the UniRecord object's data value as a `UniDynArray` object.

public UniDynArray RecordID {get; set}

This property gets or sets the `UniRecord` object's record ID as a `UniDynArray` object.

public int RecordReturnValue {get; set}

This property returns an integer representing the `UniRecord` object's return value.

public int RecordStatus {get; set}

This property gets or sets an integer representing the `UniRecord` object's status.

UniRecord – public instance methods

This section describes the protected instance methods you can use with `UniRecord` objects.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public override string ToString ( )

This method returns the `UniRecord` object as a string object. The record ID and the record's data value are combined, separated by an item mark.

This method overrides the `ToString` method of the `UniDynArray` class.

UniRecord – protected instance methods

This section describes the protected instance methods you can use with `UniRecord` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

UniSelectList class

The `UniSelectList` object lets you manipulate a select list on the server.

Select lists are described in The database environment and Select lists.

UniSelectList – public instance properties

This section describes the public instance properties you can use with `UniSelectList` objects.

public bool LastRecordRead {get;}

This property gets the status of the last record read.

UniSelectList – public instance methods

This section describes the public instance methods you can use with `UniSelectList` objects.

public void ClearList ( )

This method clears a select list, emptying the contents and preparing for a new select list to be generated.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **ClearList** method and the BASIC CLEARSELECT statement.

```
uSel.ClearList();
uSel.Select( uFile );
```

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public void FormList (string pRecID)

public void FormList (string[ ] pRecIDSet)

This method creates a select list from a supplied list of record IDs or a UniDataSet object.

`string pRecID` is a delimited string containing a list of record IDs, separated by field marks (`UniTokens.AT_FM`).

`string pRecIDSet` is a string array of record IDs.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **FormList** method and the BASIC FORMLIST statement.

```
UniDynArray testArray = testArray = new UniDynArray(uSession,"");
                for (int i = 1; i < 10; i++)
                {
                    testArray.Insert(i, "newRec" + i);
}
uSelect.ClearList();

uSelect.FormList(testArray.ToString());
```

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public void GetList (string aListName)

This method activates the named select list from the &SAVEDLISTS& file on the server.

`string aListName` is a string representing the name of the list to be activated.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **GetList** method and the database `GET.LIST` command.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public string Next ( )

This method returns the next record ID in the select list. If the list is exhausted, `Next ( )` returns a null value, and the `LastRecordRead` property returns true.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **Next** method and the BASIC READNEXT statement.

```
UniDynArray testArray = testArray = new UniDynArray(uSession,"");
                for (int i = 1; i < 10; i++)
                {
                    testArray.Insert(i, "newRec" + i);
}
uSelect.ClearList();

uSelect.FormList(testArray.ToString());

while (!uSelect.LastRecordRead)
                {
                    string  nextRec = uSelect.Next();
                    counter++;
}
```

public UniDynArray ReadList ( )

This method reads the entire contents of a select list and returns it all at once.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **ReadList** method and the BASIC READLIST statement.

```
UniFile uFile = uSession.CreateUniFile("FOOBAR");
UniSelectList uSelect = uSession.SelectList(1);
        uSelect.Select(uFile);
UniDynArray retList = uSelect.ReadList();
```

public void SaveList (string aListName)

This method saves the currently active select list in the &SAVEDLISTS& file with the specified name on the server.

`string aListName` is the file name of the list to be saved.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **SaveList** method and the `SAVE.LIST` command.

public void Select (UniDictionary uniFile)

public void Select (UniFile uniFile)

This method creates a select list by selecting the `UniFile` or `UniDictionary` object and generating a select list of all record IDs from that database file. The new select list overwrites any previous select list and resets the select list pointer to the first record in the list.

`UniDictionary uniFile` is the `UniDictionary` object to be selected.

`UniFile uniFile` is the `UniFile` object to be selected.

This example opens the ORDERS file, creates a select list of its record IDs, then starts to read records from the file using the select list:

```
UniFile uFile = uSession.CreateUniFile("ORDERS");
UniSelectList uSelect = uSession.SelectList(0);
```

```
        uSelect.Select(uFile);
UniDynArray uvr = uFile.Read(uSelect.Next());
```

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **Select** method, the BASIC SELECT statement, and the database SELECT command.

Note: `TheSelect()` method does not correspond to the SQL SELECT statement.

public void SelectAlternateKey (UniDictionary unid, string aIndexName)

public void SelectAlternateKey (UniFile uniFile, string aIndexName)

This method creates a select list from the specified `UniDictionary` or `UniFile` object from values in the specified secondary index.

`UniFile uniFile` is the `UniFile` object to be selected.

`UniDictionary uniFile` is the `UniDictionary` object to be selected.

`string aIndexName` is the name of a secondary index as specified in a database `CREATE.INDEX` command.

If the named secondary index does not exist, the select list is empty. The new select list overwrites any previous select list and resets the select list pointer to the first record in the list.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **SelectAlternateKey** method and the BASIC SELECTINDEX statement.

```
uSel.SelectAlternateKey( custFile, "CUST.ORDER.NO" );
```

public void SelectMatchingAK (UniDictionary unid, string aIndexName, string aIndexValue)

public void SelectMatchingAK (UniFile uniFile, string aIndexName, string aIndexValue)

This method creates a select list from a specified UniData or UniVerse file from record IDs whose value matches that in a named secondary index field. The select list contains record IDs.

`UniDictionary unid` is the name of the UniData or UniVerse dictionary file for which the select list is to be created.

`UniFile uniFile` is the name of the UniData or UniVerse file for which the select list is to be created.

`string aIndexName` is the name of a secondary index as specified in a database `CREATE.INDEX` command. If the index you specify does not exist, an empty select list is returned and the `LastRecordRead` property returns true.

`string aIndexValue` is a value from the secondary index. Records are selected when `aIndexValue` matches the value of the indexed field. It is equivalent to the following database `SELECT` command:

`SELECT` *`filename`* `WITH` *`indexname`* `=` *`indexvalue`*

The new select list overwrites any previous select list and resets the select list pointer to the first record in the list.

If this method fails, it throws a `UniSelectListException`.

This method corresponds to the UniObjects **SelectMatchingAk** method and the BASIC SELECTINDEX statement.

```
uSel.SelectMatchingAK( custFile, "CUST.STATE", "MA" );
```

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniSelectList – protected instance methods

This section describes the protected instance methods you can use with `UniSelectList` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniSelectList object

```
try
{
UniFile uFile = uSession.Open ("DATAFILE");
UniSelectList uSelect = uSession.SelectList (0);
UniDynArray uvr;
uSelect.Select (uFile);
uvr = uFile.Read (uSelect.Next ());
while (!uSelect.LastRecordRead)
{
uvr = uFile.Read (uSelect.Next());
//<...process record...>
}
}
catch (UniSelectListException)
{
/*deal with exception */
}
```

UniSequentialFile class

The `UniSequentialFile` object defines and manages files that are processed sequentially. A sequential file is an operating system file on the server containing text or binary data that you want to use in your application. In UniVerse, sequential files are defined as type 1 or type 19 files.

For more information about using the UniSequentialFile object, see Using binary and text files. For a program example that uses the `UniSequentialFile` object, see Example using the UniSequentialFile object.

UniSequentialFile – public instance properties

This section describes the public instance properties you can use with `UniSequentialFile` objects.

public int EncryptionType {get; set;}

This property gets or sets the type of encryption to use for all operations on `UniSequentialFile` objects.

`int` is the token number for the encryption type, as follows:

Token number	Token	Description
0	UniObjectsTokens.NO_ENCRYPT	Do not encrypt data. This is the default value.
1	UniObjectsTokens.UV_ENCRYPT	Encrypt data using internal database encryption.

public bool IsFileOpen {get;}

This property checks to see if a file is open. It returns true if file is open, or false if the file is closed.

For example:

```
UniSession us=null;
                try
                {
us =
UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");

UniSequentialFile fl =
us.CreateSequentialFile("BP",”OLDTEST”,true);

}
                catch(Exception e)
                {
                    Console.WriteLine(e.Message +e.StackTrace);
                }
                finally
                {
                    if(us != null && us.IsActive)
                    {
                        UniObjects.CloseSession(us);
                    }
```

```
}
```

This property corresponds to the UniObjects **IsOpen** method.

public bool ReadSize {get; set}

This method gets or sets the number of bytes to be read for each successive call to the `ReadBlk ( )` method.

The `ReadSize` value is initially set to 0, which indicates that all the data should be read in a single block. When the `ReadBlk ( )` method finishes, the `ReadSize` value is reset to the number of bytes that were actually read. 0 indicates an error or the end of the file.

`int` is the number of bytes to be read in one operation.

Set the value to a suitable number of bytes for the memory available to your application. Values less than 0 are treated as 0.

This method corresponds to the UniObjects **ReadSize** property.

Warning: If the value is set to 0 and there is not enough memory to hold all the data, a runtime exception occurs.

Note: Use the `ReadSize` property before each use of the `ReadBlk ( )` method because the `ReadSize` value may have been modified previously.

---

public int Timeout {get; set;}

This property gets or sets the length of time before the session times out during `ReadBlk ( )` operations. The remote procedure call (UniRPC) utility uses the timeout setting.

`int` is the timeout value in seconds. The default value is 300 seconds (5 minutes).

Note: If you enter a value that is too small, a running process may time out. If this occurs, an error code is returned and the connection to the server is dropped.

If this property fails, it throws a `UniSessionException`.

This property corresponds to the UniObjects **Timeout** property.

public int UniSequentialStatus {get; set}

This property gets or sets the status code of the last method performed on a `UniSequentialFile` object. Refer to each method for a description of these status values.

UniSequentialFile – public instance methods

This section describes the public instance methods you can use with `UniSequentialFile` objects.

public void Close ( )

This method closes a sequential file. It corresponds to the UniObjects **CloseSeqFile** method and the BASIC CLOSESEQ statement.

If this method fails, it throws a `UniSequentialFileException`.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public void FileSeek (int aRelPos, int aOffset)

This method moves the sequential file pointer by an offset position specified in bytes relative to the current position, to the beginning of the file, or to the end of the file.

`int aRelPos` is the token number for the pointer's relative position in a file, as follows:

Token number	Token	Description
0	UniObjectsTokens.UniT_START	The start of the file.
1	UniObjectsTokens.UniT_CURR	The current position.
2	UniObjectsTokens.UniT_END	The end of the file.

`int aOffsetis` the number of bytes before or after `aRelPos`. A negative offset moves the pointer to a position before `aRelPos`.

For example:

```
UniSequentialFile fl =
us.CreateSequentialFile("BP",”OLDTEST”,true);
fl.FileSeek(0,0);fl.WriteEOF();
int p = Console.Read();
fl.FileSeek(0,0); /* position back to the beginning of file */
```

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **FileSeek** method and the BASIC SEEK statement.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void Open ( )

This method opens a server-side file for sequential processing, or creates a file if the CreateFlag is set and a file does not exist.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **Open** method and the BASIC OPENSEQ statement.

public UniDynArray ReadBlk ( )

This method reads a block of data from a sequential file. The size of the data block is specified by the `ReadSize` property.

Upon completion, you can use the `ReadSize` property to determine the number of bytes read. Additionally, the `UniSequentialFileStatus` property returns a status value, as follows:

Value	Description
–1	The file is not open for a read.
0	The read was successful.
1	The end of the file was reached.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **ReadBlk** method and the BASIC READBLK statement.

```
UniSequentialFile fl =
us.CreateSequentialFile("BP",”OLDTEST”,false);
fl.ReadSize = 4096;
UniDynArray ur = fl.ReadBlk();
Console.WriteLine( "Number of bytes read " + fl.ReadSize);
Console.WriteLine ( "Status from readblk " +
uSeq.UniSequentialStatus );
```

public UniDynArray ReadLine ( )

This method reads successive lines of data from the current position in a sequential file. The lines must be delimited by an end-of-line character such as a carriage return.

Upon completion, the `UniSequentialFileStatus` property returns a status value, as follows:

Value	Description
–1	The file is not open for a read.
0	The read was successful.
1	The end of the file was reached, or the read-size value is 0 or less.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **ReadLine** method and the BASIC READSEQ statement.

```
UniSequentialFile fl =
us.CreateSequentialFile("BP",”OLDTEST”,false);
UniSequentialFile fl2 =
us.CreateSequentialFile("BP",”OLDTEST2”,true);

UniDynArray uvstr = fl.ReadLine();
int uvstat = fl.UniSequentialStatus;
while ( uvstat == 0 )
{
uvstr = uSeq.ReadLine();
fl2.WriteLine(uvstr);
uvstat = fl.UniSequentialStatus;
```

```
}
```

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

public void WriteBlk (UniDynArray aString)

public void WriteBlk (string aString)

This method writes a block of data at the current position in a sequential file.

`UniDynArray aString` or `string aString` is the block of data to be written to the file.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **WriteBlk** method and the BASIC WRITEBLK statement.

```
UniSequentialFile fl =
us.CreateSequentialFile("BP",”OLDTEST”,true);
fl.ReadSize = 4096;
UniDynArray ur = fl.ReadBlk();
UniSequentialFile uSeq = uSession.openSeq("BP", "TEST”,true);
fl.WriteBlk(ur);
```

public void WriteEOF ( )

This method writes an end-of-file marker at the current position in the sequential file. This allows a file to be truncated at a specified point when used with the `FileSeek` ( ) method.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **WriteEOF** method and the BASIC WEOFSEQ statement.

public void WriteLine (UniDynArray aString)

public void WriteLine (string aString)

This method writes a line of data at the current position in the sequential file.

`UniDynArray aString` or `string aString` is a line of data to be written to the file.

If this method fails, it throws a `UniSequentialFileException`.

This method corresponds to the UniObjects **WriteLine** method and the BASIC WRITESEQ statement.

UniSequentialFile – protected instance methods

This section lists the protected instance methods you can use with `UniSequentialFile` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniSequentialFile object

```
UniSequentialFile uSeq = uSession.Create UniSequentialFile ( "BP",
"TEST2",false);
Console.WriteLine ( "Opened file" );
if ( uSeq.IsOpen )
{
Console.WriteLine ( "Setting new block size" );
uSeq.ReadSize =  4096;
Console.WriteLine ( "Reading blk from file " );
UniDynArray uvstr = uSeq.readBlk();
Console.WriteLine ( "Displaying blk from file" );
Console.WriteLine ( "Number of bytes read " + uSeq.ReadSize);
Console.WriteLine ( "Status from readblk " + uSeq. UniSequentialStatus );
Console.WriteLine ( uvstr );
// Let's open up a new sequential file
UniSequentialFile uSeqnew = uSession.openSeq( "BP", "TEST.JAVA2", true );
uSeqnew.WriteBlk( uvstr );
uSeqnew.FileSeek( 0,0 );
uSeqnew.WriteEOF();
Console.WriteLine ("Press Return to continue");
Int myinput = Console.Read();
Console.WriteLine ();
Console.WriteLine ( "Ok, let's reset the filepointer  back to 0" );
uSeq.FileSeek( 0,0 );/* Position back to beginning offile */
Console.WriteLine ( "Let's read a line at time" );
uvstr = uSeq.ReadLine();
Console.WriteLine ( "First line = " + uvstr );
int uvstat = uSeq. UniSequentialStatus ;
/productinfo/alldoc/UNIVERSE10/java/Ch4
2/11/02
Console.WriteLine ( "Status after first read = " + uvstat );
while ( uvstat == 0 )
{
uvstr = uSeq.ReadLine();
uSeqnew.WriteLine( uvstr );
uvstat = uSeq. UniSequentialStatus ;
Console.WriteLine ( "Line = " + uvstr );
}
Console.WriteLine ( "Final Status = " + uvstat );
uSeq.Close();
uSeqnew.Close();
}
Console.WriteLine ( "Closing session " );
UniObjects.CloseSession( uSession );
}
catch (Exception e)
{
                    Console.WriteLine(e.Message +e.StackTrace);
```

```
}
}
```

UniSubroutine class

The `UniSubroutine` class allows users to run a cataloged BASIC subroutine on the server.

For information about subroutines, see Client/server design considerations.

UniSubroutine – public instance properties

This section describes the public instance properties you can use with `UniSubroutine` objects.

public int ArgumentsNumber {get;}

This property gets the number of arguments this subroutine expects to use.

public string RoutineName {get;}

This property gets the name of the subroutine to call on the server. It corresponds to the UniObjects **RoutineName** property.

UniSubroutine – public instance methods

This section describes the public instance methods you can use with `UniSubroutine` objects.

public void Call ( )

This method executes the cataloged UniData or UniVerse subroutine identified by the `RoutineName` property, or identified in the `UniSession.CreateUniSubroutine` ( ) call. It uses the arguments established with the `SetArg`() method.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **Call** method and the BASIC CALL statement.

```
UniSubroutine uSub = uSession.CreateUniSubroutine("SAMPLESUBR",
3);
uSub.SetArg(0, "David");
uSub.SetArg(1, "Thomas");
uSub.SetArg(2, "Meeks");
uSub.Call();
```

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public string GetArg (int aArgNum)

This method retrieves argument values returned from the subroutine after the `Call` ( )method has executed successfully.

`int aArgNum` is the number of the argument you are requesting. The first argument is 0.

If this method fails, it throws a `UniFileException`.

This method corresponds to the UniObjects **GetArg** method.

public UniDynArray GetArgDynArray (int aArgNum)

This method retrieves argument values returned from the subroutine after the `Call` ( )method has executed successfully, as a UniDynArray.

`int aArgNum` is the number of the argument you are requesting. The first argument is 0.

If this method fails, it throws a `UniFileException`.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public void ResetArgs ( )

This method resets the output argument array of the UniSubroutine object to empty values. It corresponds to the UniObjects **ResetArgs** method.

public void SetArg (int aArgNum, string aArgVal)

public void SetArg (int aArgNum, UniDynArray aArgVal)

This method sets the value of an argument to be passed to a cataloged subroutine.

`int aArgNum` is the number of the argument you are setting. The first argument is 0.

`string aArgVal` is the value of the argument to pass to the server subroutine.

`UniDynArray aArgVal` is a UniDynArray representing the value of the argument to pass to the server subroutine.

The argument is passed to the server before making the call. Any argument you do not specify with the `SetArg` ( ) method is passed as an empty string.

If this method fails, it throws a `UniSubroutineException`.

This method corresponds to the UniObjects **SetArg** method.

```
UniSubroutine uSub = uSession.CreateUniSubroutine("SAMPLESUBR",
3);
uSub.SetArg(0, "David");
uSub.SetArg(1, "Thomas");
uSub.SetArg(2, "Meeks");
uSub.Call();
```

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniSubroutine – protected instance methods

This section describes the protected instance methods you can use with `UniSubroutine` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniSubroutine object

```
UniSubroutine uSub = uSession. CreateUniSubroutine
("SAMPLESUBR",3);
uSub.SetArg(0, "David");
uSub.SetArg(1, "Thomas");
uSub.SetArg(2, "Meeks");

Console.WriteLine("Subroutine set up, routine name = " +
uSub.RoutineName);

Console.WriteLine ("Subroutine: Arg0 = " + uSub.GetArg(0));
Console.WriteLine (" Arg1 = " + uSub.GetArg(1));
Console.WriteLine (" Arg2 = " +uSub.GetArg(2));
Console.WriteLine ("Calling subroutine...");

uSub.Call();

Console.WriteLine ("Subroutine finished... ");
Console.WriteLine ("Subroutine: Arg0 = " + uSub.GetArg(0));
Console.WriteLine (" Arg1 = " + uSub.GetArg(1));
Console.WriteLine (" Arg2 = " + uSub.GetArg(2));
```

```
Console.WriteLine ("Results displayed, resetting args...");

uSub.ResetArgs();
Console.WriteLine ("Subroutine: Arg0 = " +uSub.GetArg(0));
Console.WriteLine (" Arg1 = " +uSub.GetArg(1));
Console.WriteLine (" Arg2 = " + uSub.GetArg(2));
```

UniTransaction class

The `UniTransaction` object is available from the `UniSession` object. The `UniTransaction` class provides methods to start, commit, and roll back transactions for a session. If a session closes while transactions are active, the server rolls them back. For any `UniSession` object, only one transaction can be active at a time.

UniTransaction – public instance methods

This section describes the public instance methods you can use with `UniTransaction` objects.

public void Begin ( )

This method begins a new transaction. This transaction can be nested. If a transaction is already active, the nested transaction becomes active and the transaction level is incremented.

If this method fails, it throws a `UniTransactionException`.

This method corresponds to the UniObjects **Start** method and the BASIC BEGIN TRANSACTION statement.

public void Commit ( )

This method commits an active transaction. If it is a nested transaction, the parent transaction becomes active and the transaction level is decremented.

If this method fails, it throws a `UniTransactionException`.

This method corresponds to the UniObjects **Commit** method and the BASIC COMMIT statement.

public void Dispose( )

This method is inherited from `UniRoot`. It performs cleanup for the session.

public static bool Equals (object, object)

This method is inherited from `Object`. It is used to determine whether the specified object is equal to the current object.

public virtual int GetHashCode( )

This method is inherited from `Object`. It serves as a hash function for a particular type. It is best suited for use in hashing algorithms and data structures, such as hash tables.

public int GetLevel ( )

This method returns the current transaction level. It corresponds to the UniObjects **Level** property.

> **Note:** This method applies only to UniVerse.

If this method fails, it throws a `UniTransactionException`.

public Type GetType( )

This method is inherited from `Object`. It gets the type of the current instance.

public bool IsActive ( )

This method determines whether a transaction is currently active. It returns `true` if the transaction is active, otherwise it returns `false`. A transaction is currently active if the `UniTransaction.Begin` ( ) method has been called, but neither `UniTransaction.Commit` ( ) nor `UniTransaction.Rollback` ( ) has been called.

If this method fails, it throws a `UniTransactionException`.

This method corresponds to the UniObjects **IsActive** method.

public void Rollback ( )

This method rolls back an active transaction. If this is a nested transaction, the parent transaction becomes active and the transaction level is decremented.

If this method fails, it throws a `UniTransactionException`.

This method corresponds to the UniObjects **Rollback** method and the BASIC ROLLBACK statement.

public virtual string ToString( )

This method is inherited from `Object`. It returns a string that represents the current object.

UniTransaction – protected instance methods

This section lists the protected instance methods you can use with `UniTransaction` objects.

protected override void Dispose (bool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize( )

This method is inherited from `Object`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone( )

This method is inherited from `Object`. It creates a shallow copy of the current object.

Example using the UniTransaction object

```
UniSession uSession =null;

Try
{

 uSession = UniObjects.OpenSession("localhost","xxx","yyy","demo","udcs");

UniTransaction uvt = uSession.CreateUniTransaction();
/* Ok, let's open up a file and first write a record outside the
transaction  */
UniFile uFile = uSession.CreateUniFile("CUSTOMER");
UniDynArray uvstr = uFile.Read("2");
UniDynArray uvnewstr = new UniDynArray(uSession,"This is a test of
Transactions 1 ");
uFile.Write("TRANSREC", uvnewstr.StringValue);

Console.WriteLine("Data written outside transaction... check on  it ");
Int lval = Console.Read();
Char ch = (char)lval;
Console.WriteLine ("Starting transaction");
Console.WriteLine ("Current transLevel = " + uvt.GetLevel());
Console.WriteLine ("Is it active? " + uvt.IsActive());
uvt.Rollback();
uvt.Begin();
Console.WriteLine ("Transaction started: Level " +  uvt.GetLevel());
uFile.Write("TRANSCOMMITREC", uvnewstr.StringValue);
Console.WriteLine ("Data written, but not committed... hit any  key to
continue");
lval = Console.Read();
ch = (char)lval;
if (ch  == 'Y')
{
uvt.Commit();
Console.WriteLine ("Commited task... hit any key to  continue");
lval = Console.Read();
ch = (char)lval;
}
else
{
uvt.Rollback();
Console.WriteLine ("Rolledback... hit any key to continue");
lval = Console.Read();
ch = (char)lval;}
Console.WriteLine ("Closing session");
                }

                Catch (Exception ex)
                        {
                        //some error, display it
                        Console.WriteLine(ex.Message);
                        }
                        finally
                        {
                        // no error
                        if(uSession!= null)
                        {
                        UniObjects.CloseSession(uSession);
                        uSession = null;
                        }
```

```
}
```

UniXML class

The UniXML class represents an XML representation of UniData data. Using this class, you can create XML documents and XML Schema documents from UniQuery or UniData SQL, or directly from a data file. UniData also provides functions to generate new data, modify data, or generate XML from the UniData database using the XMAP file.

UniXML – public instance properties

This section describes the public instance properties you can use with UniXML objects.

public int Errcode {get;}

This property gets a UniXML error code.

public string Errmsg {get;}

This property gets a UniXML error message.

public string XMLString {get; set;}

This property gets or sets an XML document as a string type.

public string XSDString {get; set;}

This property gets or sets an XML schema as a string type.

UniXML – public instance methods

This section describes the public methods you can use the UniXML objects.

public void GenerateXML(string cmd);

This method uses the UniQuery `LIST` command or the UniData SQL `SELECT` command to get an XML document from the UniData server. If you only supply a command, UniObjects for .NET sets the option to an empty string, checks the result, and reports an error if one occurs.

public void GenerateXML(string cmd, string options);

This method uses the UniQuery LIST command or the UniData SELECT command to get an XML document from the UniData server. You can specify options separated by @FM, and option values separated by @VM. The XMLExecute options are as follows:

Description	
WITHDTD	Creates a DTD and binds it with the XML document. By default, UniVerse creates an XML schema. However, if you include WITHDTD in your RetrieVe or UniVerse SQL statement, UniVerse does not create an XML schema, but only produces the DTD.
ELEMENTS	The XML output is in element-centric format.
'XMLMAPPING': @VM:'*mapping_file_ name*'	Specifies the mapping file containing transformation rules for display. This file must exist in the &XML& directory.
'SCHEMA':@VM: '*type*'	The default schema format is ref type schema. You can use the SCHEMA attribute to define a different schema format.
HIDEMV, HIDEMS	Normally, when UniVerse processes multivalued or multi-subvalued fields, UniVerse adds another level of elements to produce multiple levels of nesting. You have the option of disabling this additional level by adding the HIDEMV and HIDEMS attributes. When these options are on, the generated XML document and the associated DTD or XML schema have fewer levels of nesting.
HIDEROOT	Allows you to specify to only create a segment of an XML document, for example, using the SAMPLE keyword and other conditional clauses. If you specify HIDEROOT, UniVerse only creates the record portion of the XML document, it does not create a DTD or XML schema.
'RECORD':@VM: '*newrecords*'	The default record name is FILENAME_*record*. The record attribute in the ROOT element changes the record name.
'ROOT':@VM: '*newroot*'	The default root element name in an XML document is ROOT. You can change the name of the root element as shown in the following example: root="*root_element_name*"
TARGETNAME-SPACE:@FM:'*namespaceURL*'	UniVerse displays the targetnamespace attribute in the XMLSchema as *targetNamespace*, and uses the URL you specify to define *schemaLocation*. If you define the targetnamespace and other explicit namespace definitions, UniVerse checks if the explicitly defined namespace has the same URL and the targetnamespace. If it does, UniVerse uses the namespace name to qualify the schema element, and the XML document element name.
COLLAPSEMV, COLLAPSEMS	Normally, when UniVerse processes multivalued or multi-subvalued fields, UniVerse adds another level of elements to produce multiple levels of nesting. You have the option of disabling this additional level by adding the COLLAPSEMV and COLLAPSE MS attributes. When these options are on, the generated XML document and the associated DTD or XML Schema have fewer levels of nesting.

public void GenerateXMLUsingXmap(string xmapname);

This method uses an existing XMAP file on the server to generate an XML document from UniData data. The relationship between an XML document and a UniData file is described in the XMAP file. The XML document is returned as a string.

public DataSet GetDataSet();

This method returns a DataSet using m_Xmlstr and m_Xsdstr if one exists.

public void UpdataDataUsingXmap(string xmapname);

This method writes to a UniData file residing on the server using an existing XMAP file and the m_Xmlstr residing on the client. The XMAP file is stored in the UniData account in the _XML_ file.

public void UpdataDataUsingXmap(string xmapname, string xmlname);

This method writes a UniData file residing on the server using the XMAP file residing on the server and an XML document residing on the server.

UniXML – protected instance methods

This section describes the protected instance methods you can use with the UniXML object.

protected override void Dispose (pool disposing)

This method is inherited from `UniRoot`. It overrides the `Dispose`() method.

protected Finalize()

This method is inherited from `UniRoot`. It allows an object to attempt to free resources and perform other cleanup operations before the object is reclaimed by garbage collection.

protected object MemberwiseClone()

This method is inherited from `Object`. It creates a shallow copy of the current object.

Chapter 4: Getting started with UniObjects for .NET

Setting up UniObjects for .NET

This section contains information on the requirements for setting up and installing UniObjects for .NET in your environment:

- Software requirements, on page 147
- Hardware requirements, on page 147
- Installing UniObjects for .NET, on page 148

Software requirements

This section lists the software required to support UniObjects for .NET. Both client and documentation software components are required.

Client software components

The following table lists the client software components that must be installed before you install UniObjects for .NET.

Software	Requirement
Operating system	Windows 8.1 (or later)
.NET Framework	Version 4.5 or later
UniData	Version 7.2 or later
UniVerse	Version 11.1 or later

Documentation software components

The following table lists the documentation software components that must be installed on your computer to support UniObjects for .NET.

Software	Requirement
Internet software	Microsoft Internet Explorer version 5.01 or later

Hardware requirements

The following table lists the hardware required to support UniObjects for .NET.

Software	Requirement
Processor	Pentium 450 megahertz (MHz) minimum; Pentium 733 MHz or greater recommended

Software	Requirement
Memory	128 megabytes (MB) RAM minimum; 256 MB RAM recommended
Hard disk space	UniObjects for .NET component: 1 MB

Installing UniObjects for .NET

UniData 7.1 offers several options for installing UniObjects for .NET:

- Installing with the InstallShield Wizard, on page 148
- Installing from the Control Panel, on page 149

Installing with the InstallShield Wizard

The easiest method to install UniObjects for .NET is to use the InstallShield Wizard, which walks you through the process. Complete the following steps to install program files with the InstallShield Wizard:

1. Locate the Install.exe file in the installation directory on the U2 Client CD. Double-click the file to open it.
   The UniData installation menu appears.
2. On the installation menu, click **UniDK** to install the Uni Development Kit.
   This starts the InstallShield Wizard for UniDK.
3. To continue the installation, click **Next**.
4. To install UniObjects for .NET to the default destination folder, click **Next**.
   Otherwise, to install to a different folder, click **Browse** to locate a folder and then click **Next**.
5. The **UniObjects for .NET** check box is selected by default. Clear the check box for any other components you do not want to install.
6. Click **Next**. By default, the installation program does not overwrite the uci.config and uvodbc.config files from a previous installation of UniDK.
   If you want to overwrite existing files from a previous version of UniDK, select the **Overwrite Preserved Files** check box.
7. Click **Next**. Select the program folder to which Setup will add the UniObjects for .NET program icon.
   To accept the default program folder, click **Next**.
   Otherwise, type a new folder name or select one from the **Existing Folders** list, and then click **Next**.
   The Start Copying Files dialog box appears.
8. If the current settings are satisfactory, click **Next** to start copying the program files.
   Otherwise, if you need to change a setting, click the **Back** button to return to the dialog box in which you want to make a change.
   The Setup Status dialog box appears, showing the percentage of files copied.
   When all files are copied, the Setup Complete dialog box appears.
9. The **View Release Notes** check box is selected by default. If you do not want to view release notes, clear the check box.
10. Click Finish. If the **View Release Notes** check box was cleared, the InstallShield Wizard Complete dialog box appears.
    If the View Release notes check box was selected, the release notes file opens. After reading the release notes, close the file. The InstallShield Wizard Complete dialog box appears.
11. Select the **Yes** option to restart your computer now or the **No** option to restart later.
12. Click **Finish**.

Installing from the Control Panel

Complete the following steps to install UniObjects for .NET from the Control Panel:

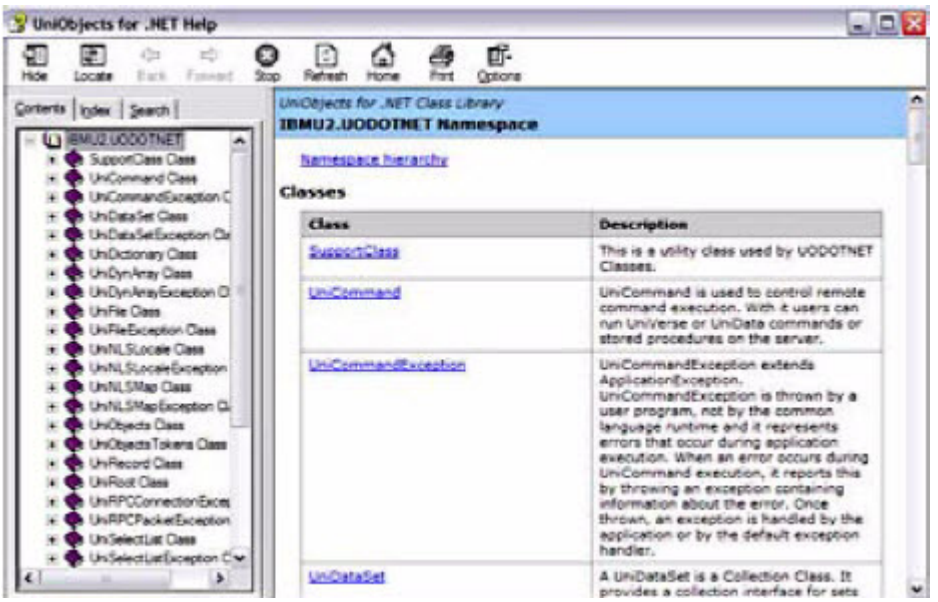
1. Select **Start > Settings > Control Panel > Add/Remove Programs**.
2. Click **Add New Programs**.
3. Click the **CD or Floppy** button.
4. Insert the UniVerse 10.2 Product CD in your CD drive.
5. Click **Next**.
6. Click the **Browse** button to locate the `uosetup.msi` file in the installation directory on the product CD.

Using online help

This section describes how you can use online help to get information UniObjects for .NET.

Online help for UniObjects for .NET is available in the `uodotnet.chm` file. This file is found in the installation directory (generally the `UONET\DOC` directory). To view the online help, double-click the `uodotnet.chm` file to open it.

The online help is modeled after MDSN Web help, so it has the same look and feel. A sample of the help window is shown below:



Deploying .NET applications

After creating a software application or Web service using the UniObjects for .NET API, the next step is to deploy the product to users. Developers who use the UniObjects for .NET component in their applications can include the `UOMergeModule.msm` merge module in their `.msi` files to ensure that the UniObjects for .NET component is installed correctly along with the other application files. This file is provided in the `UONET\UOMergeModule` directory on the UniData product CD.

Merge modules are a feature of Windows Installer that provides a standard method for delivering components, ensuring that the correct version of a component is installed. A merge module contains a component such as a .dll along with any related files, resources, registry entries, and setup logic.

Merge modules cannot be installed directly, but rather are merged into an installer for each application that uses the component. This ensures that the component is installed consistently for all applications, eliminating problems such as version conflicts, missing registry entries, and improperly installed files.

You can ensure that an assembly name is globally unique by signing it with a strong name. The UniObjects for .NET assembly is signed with a strong name using the key file `uodotnet.snk`. This file is provided in the `UONET\BIN` directory on the UniData product CD.

Strong names offer:

- Guarantee that the name is unique by relying on unique key pairs. No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
- Protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
- Provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that is provided by a digital signature and supporting certificate, for example.

Chapter 5: Getting started with UniObjects for .NET compact framework

This chapter introduces the UniObjects for .NET Compact Framework interface. The chapter begins with a brief exploration of the .NET Compact Framework environment and how that is used in UniObjects for .NET. It goes on to explain the differences between UniObjects for .NET and UniObjects for .NET Compact Framework. The final section demonstrates how to build a UniObjects for .NET Compact Framework application in Visual Studio 2005.

About UniObjects for .NET compact framework

UniObjects for .NET Compact Framework is a proprietary middle-ware application program interface (API) designed specifically for software development in the .NET Compact Framework. This interface is managed code written in C#.

Software developers can use the UniObjects for .NET Compact Framework API within Visual Studio, and any CLR language (such as C#, VB.NET, or C++) to create applications for small handheld devices.

You will need to know more about the .NET Compact Framework before we go on to discuss its use with UniObjects. The next section gives you a brief introduction to the .NET Compact Framework. For more information, please refer to Getting started with UniObjects for .NET, on page 147.

About the .NET compact framework

The .NET Compact Framework is a subset of the .NET Framework, which is a complete software package for developing and delivering software applications.

The .NET Compact Framework is an environment in which managed applications are designed to run on Windows CE based devices, such as PDAs, mobile phones, factory controllers, and set-top boxes.

For a complete discussion on the differences between the .NET Framework and the .NET Compact Framework, please visit the MSDN Web site at http://msdn.microsoft.com.

What is the .NET compact framework?

The .NET Compact Framework is an integral Windows component for building and running software applications and Web services on small hand-held devices. It is composed of an optimized Common Language Runtime (CLR) and a unified set of class libraries.

The .NET Compact Framework is specifically designed to work with the limited resources available in the Windows CE Framework. It inherits the .NET system architecture and can use approximately 30 percent of the full .NET Framework class library. The .NET Compact Framework also has several libraries designed specifically for mobile devices.

Windows CE

Windows CE is the component-based, embedded operating system used by the .NET Compact Framework. The Windows CE system was created to work specifically with intelligent devices, and allows desktop developers to smoothly move desktop applications to these devices. Users can build an application that includes features such as forms, graphics, and Web services using a handheld device.

Common Language Runtime (CLR)

The CLR is responsible for run-time services such as

- language integration
- security enforcement
- memory, process, and thread management

In addition, CLR has a role at development time when features such as lifecycle management, strong type naming, cross-language exception handling, and dynamic binding reduce the amount of code that the software developer must write to turn business logic into a reusable component.

Microsoft Intermediate Language (MSIL)

The .NET Framework compilers generate this CPU-independent instruction set for the use of the Common Language Runtime. Before MSIL can be executed, the CLR must convert it to native, CPU-specific code.

Managed code

This type of code is executed and managed by the .NET Framework's Common Language Runtime. Managed code must supply the instruction set necessary for the CLR to provide services such as memory management, cross-language integration, code access security, and automatic lifetime control of objects. All code that has been compiled in Microsoft Intermediate Language executes as managed code.

Unmanaged code

This type of code is executed by the operating system, outside the .NET Framework's Common Language Runtime. Unmanaged code must provide its own memory management, type checking, and security support, unlike managed code, which receives these services from the Common Language Runtime.

Class libraries

The .NET Framework uses a number of class libraries, listed below. Together, the class libraries provide a common, consistent development interface across all languages supported by the .NET Framework.

- **Base classes** provide standard functionality such as input/output, string manipulation, security management, network communications, thread management, text management, and user interface design features.
- **ADO.NET** classes enable developers to interact with data accessed in the form of XML through the OLE DB, ODBC, Oracle, and SQL Server interfaces.
- **XML classes** enable XML manipulation, searching, and translation.
- **ASP.NET** classes support the development of Web-based applications and Web services.
- **Windows Forms** classes support the development of desktop-based smart client applications.

IBMU2.UODOTNET is the namespace assigned to UniObjects for .NET Compact Framework for the UniData and UniVerse databases.

UniObjects for .NET Compact Framework is the data access model for .NET Compact Framework applications that connect to the UniData and UniVerse databases. It contains a collection of classes that allow you to connect to the UniData and UniVerse databases, execute commands, and read and write results:

- The UniSession class represents an open session to a UniData or UniVerse database.
- The UniFile and UniDictionary classes are used to access all file operations.
- The UniCommand class controls execution of database commands on the server. With it, users can run UniData or UniVerse commands or stored procedures on the server.
- The UniSubroutine class is used to execute cataloged BASIC server-side subroutine commands on the server.
- The UniTransaction class represents a Basic transaction to be made in a UniData or UniVerse database.
- The UniDataSet is a collection class used to read and write bulk UniRecord transactions in a UniData or UniVerse database.

Features of UniObjects for .NET compact framework

UniObjects for .NET Compact Framework is managed code, written purely in C#. It does not use any functions outside of the .NET Compact Framework optimized CLR. The UniObjects for .NET Compact Framework code complies with the .NET Framework standard and it follows C# conventions for names, comments, and standards.

UniObjects for .NET Compact Framework shares the same source code base with UniObjects for .NET, and as such, most of the information in this guide applies to both UniObjects for .NET and UniObjects for .NET Compact Framework.

Limitations of UniObjects for .NET compact framework

UniObjects for .NET and UniObjects for .NET Compact Framework share most of their core functions. There are, however, a few functional exceptions to UniObjects for .NET Compact Framework due to the constraints of the Compact Framework.

At this time, UniObjects for .NET Compact Framework does not support the following:

- Connection pooling
- Tracing and logging
- Performance counters
- Secure connection
- Configuration files

The SSL Stream class is not yet available in the .NET Compact Framework. Therefore, to ensure a secure connection for smart devices used out side a company's Intranet, we recommended that you obtain a secure connection at a lower network level by using a VPN connection from the smart device to the company's network.

Server-side applications still have logging and tracing capabilities, however, logging and tracing capabilities are not available on client-side applications at this release.

Setting up the development environment in UniObjects for .NET compact framework

This section contains information on the requirements for setting up and installing UniObjects for .NET Compact Framework in your development environment:

Software requirements

This section lists the software required to support UniObjects for .NET Compact Framework. Both client and documentation software components are required.

Client software components

The following table lists the client software components that must be installed before you install UniObjects for .NET Compact Framework.

Software	Requirement
Operating system	Windows 8.1 (or later)
.NET Framework	Version 4.5 or later
UniData	Version 7.2 or later
UniVerse	Version 11.1 or later

Documentation software components

The following table lists the documentation software components that must be installed on your computer to support UniObjects for .NET.

Software	Requirement
Internet software	Microsoft Internet Explorer version 5.01 or later

Hardware requirements

The following table lists the development environment hardware required to support UniObjects for .NET Compact Framework.

Software	Requirement
Processor	Pentium 1 gigahertz (GHz) or greater recommended
Memory	512 MB RAM minimum
Hard disk space	UniObjects for .NET component: 1 MB UniObjects for .NET Compact Framework component: 1 MB

Mobile CE hardware requirements

The following table lists the development workstation hardware required to support UniObjects for .NET Compact Framework.

Software	Requirement
Processor	400 megahertz (MHz) minimum
Memory	128 MB ROM; 128 MB RAM minimum
Operating System	Mobile 5 or Pocket PC 2003 (or later)

Installing UniObjects for .NET compact framework

UniObjects for .NET Compact Framework is installed as part of the UniObjects Developer Kit (UniDK) installed on the U2 Clients CD. UniVerse 10.3 offers several options for installing UniObjects for .NET.

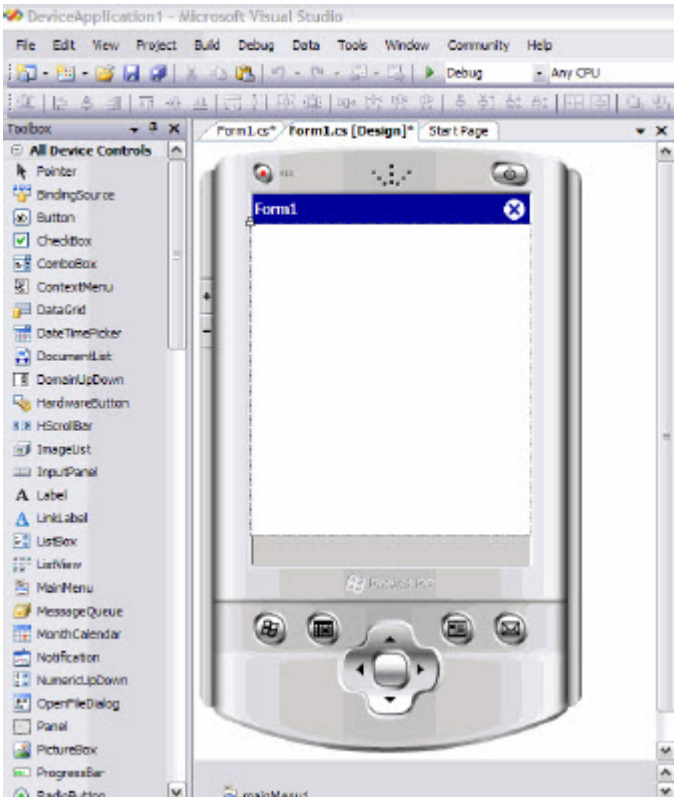
Refer to Getting started with UniObjects for .NET, on page 147 for complete installation instructions.

Creating a UniObjects for .NET compact framework application

1. Open Visual Studio 2005. Select **File > New** → **Project** from the toolbar menu. The New Project window opens.
2. Select the appropriate programming language from the project types window, and then expand **[+]** your selection.

   Note: The following example uses C#.

3. Select **Smart Device** from the list of devices, and then click the **[+]** button to expand the tree view. Select the appropriate device for your application. In this example, we use the Pocket PC 2003 device.
4. Give the application a name and then select **OK** to return focus to the Visual Studio window. An image of the selected device now appears in the design pane.



5. In the Solution Explorer pane, right-click the project name and select **Add Reference** from the menu.
   Select the **Browse** tab and navigate to the **UODOTNET4CF.dll**. The default location for this file is: `C:\U2\unidk\uonet\bin`
6. Click **OK**. Focus returns to the Visual Studio design pane.

To create an application:

Once you have created a project and added all of the appropriate references, you are ready to begin building your application.

In the following example, we create a UniXML application that displays customer information on the mobile device emulator.

1. Drag two buttons onto the form and position them at the top of the page.
2. Drag a DataGrid control onto the form and position the grid under the button controls.
3. Drag a Label onto the form and position it under the DataGrid control.
4. Drag a TextBox onto the form and position it to the right of the Label.
5. In the Properties pane, set the properties for the controls you just created as follows:

Control name	Property value
button 1 (Name)	Set the Button 1 name to **btnRUN**
button 1Text	Set the Button 1 text to **RUN**
button 2 (Name)	Set the Button 2 name to **btnQUIT**
button 2 Text	Set the Button 2 text to **QUIT**
Label Text	Set the Label text to **Customer ID**
TextBox Text	Set the TextBox Text field to **empty**

6. Once you have created all of the controls and updated all of the properties for those controls, you must create event handlers for each button control. To do this, double-click each control to open the Form.cs code view.
While in the code view, you can change the code to run the application. Use the example code below to generate a list of customer names from the CUSTOMER file, and then display them on the grid.

   Note: You must double-click each control in the following order to generate the example code: **RUN**, **QUIT**, **DataGrid**, **TextBox**.

```
...

private void btnRUN_Click(object sender, EventArgs e)
            {
                     // U2 DB Connect
                    // for UniData use ("server_name", "user_id",
"password", "demo", "udcs");
                    // UniVerse
                    UniSession us = UniObjects.OpenSession("server_name",
"user_ID", "password", "HS.SALES", "uvcs");

                    // Create XML Object
                    UniXML unixml = us.CreateUniXML();
                    String id = textBox1.Text;

                     // Create XML document from LIST Command
                    unixml.GenerateXML("LIST CUSTOMER " + id + " STATE" +
                    " CITY" + " ZIP");

                    // Get a dataset from XML
                    DataSet ds = unixml.GetDataSet();
                    dataGrid1.DataSource = ds.Tables[0];

                    // Display dataset
                    dataGrid1.Show();

                    // Close session
                    UniObjects.CloseSession(us);
                }
                private void btnQUIT_Click(object sender, EventArgs e)
                {
                    Close();
                }
                private void dataGrid1_CurrentCellChanged(object sender,
                 EventArgs e)
                {

                }

                private void textBox1_TextChanged(object sender, EventArgs
e)
                {

                }
        }
}
```

7. While in the code view, you also need to add the `IBMU2.UODOTNET` namespace to the application. Do this by adding the following code to the top of the page:

```
using IBMU2.UODOTNET;
```

Building the application

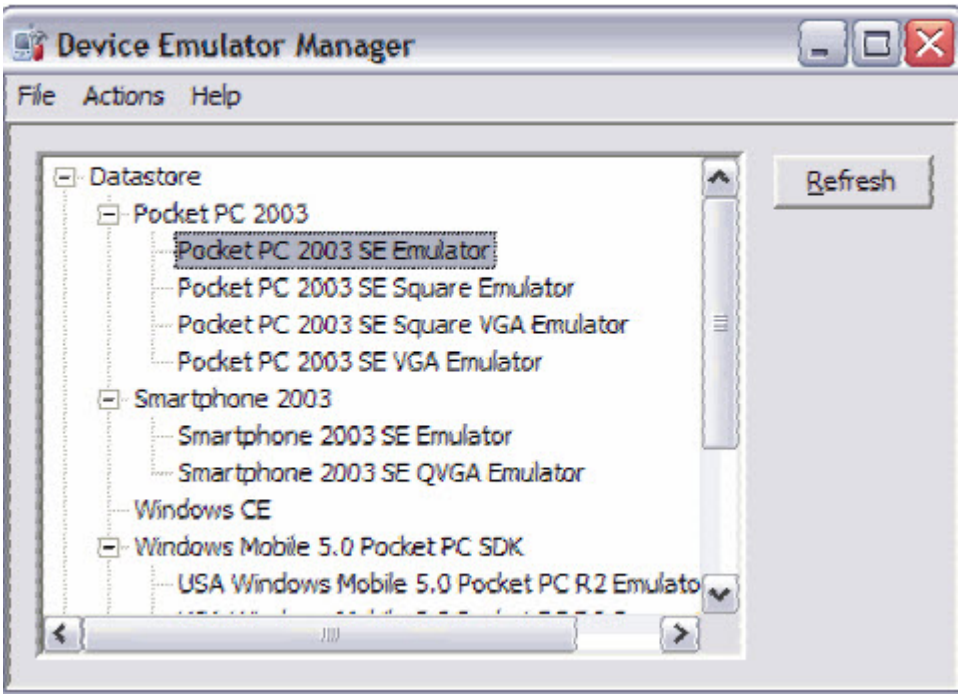
Once you have created the application, you will want to test it. Visual Studio allows you to test the application using the Device Emulator. The Emulator is a Visual Studio tool used to mimic device behavior for testing purposes, such as display orientation, serial port mapping, and networking support.

Note: You must have Microsoft ActiveSync installed in order to test the application. You can download the installation from the Microsoft Web site.

To build the application:

If it is not already open, open your application in Visual Studio 2005.

1. Build the application. To build the application, select **Build > Build Solution from the Visual Studio toolbar.**
2. Run the application. To run the application, select **Tools** → **Device Emulator Manager** from the Visual Studio toolbar. The Device Emulator Manager opens



3. A tree view list of devices appears in the Device Emulator Manager. Right-click the appropriate device and select **Connect** from the menu. The device emulator opens.
4. Click the **Refresh** button on the Device Emulator Manager. A green arrow now appears next to the selected device. Right-click the device and select **Cradle** from the menu.
5. The **ActiveSync New Partnership** wizard opens. You must use the wizard to establish a connection between your computer and the mobile device.
6. The wizard asks you to create a partnership between your computer and the emulator. Select the **Guest Partnership** option and then click **Next**.

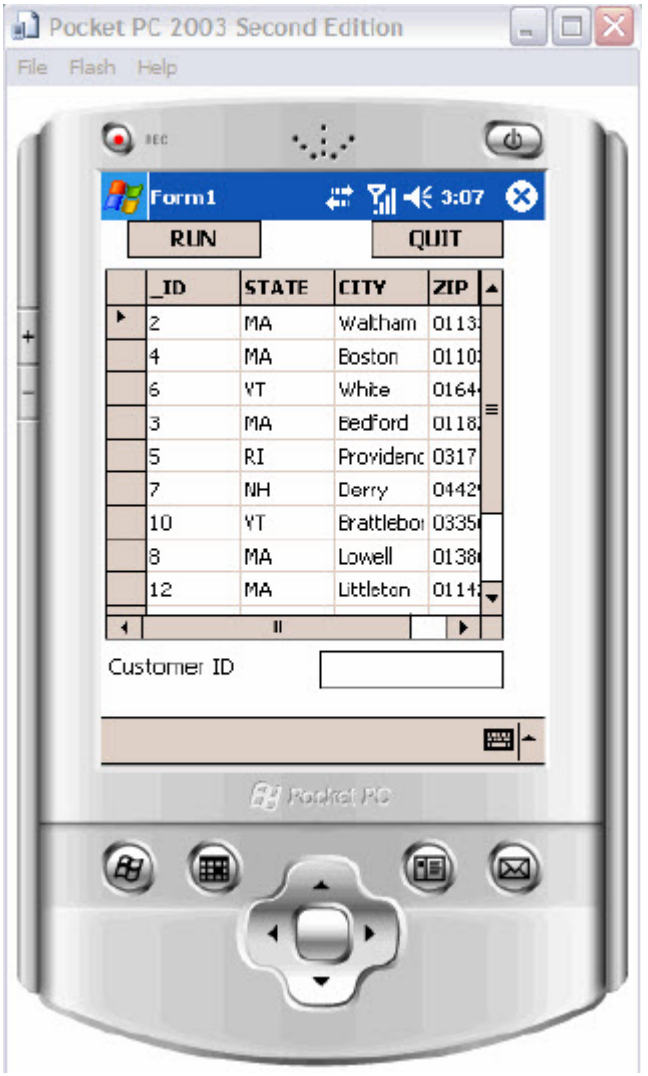
Note: If you are linking to an actual device, choose the Standard Partnership option and follow the wizard instructions.

The Microsoft ActiveSync window opens and informs you about the connection status of your application.

7. Once the connection is established, you can start the application. To start the application, select **Debug > StartWithoutDebugging** from the Visual Studio toolbar. The Deploy Device dialog box opens.
8. Choose the appropriate deployment device from the list that appears in the Deploy Device dialog box. For this example, we use the Pocket PC 2003 SE Emulator. Click **Deploy**.

The Device Emulator is now connected and ready to begin testing your application.

Click the **Run** button on your application. The information from the CUSTOMER file appears on the screen, as shown below.



Once the application is running, enter a specific customer ID into the text box and then click the **RUN** button. The information specific to that customer will display on the screen.

When you are done working with the application, click the **QUIT** button.

Chapter 6: Using TLS/SSL with UniObjects for .NET

This chapter explains how to use TLS/SSL (Secure Socket Layer) with UniObjects for .NET.

Overview of TLS/SSL technology

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are transport layer protocols that provides a secure channel between two communicating programs over which arbitrary application data can be sent securely. It is by far the most widely deployed security protocol used on the World Wide Web.

Note: TLSv1.1 is the default security protocol. We recommend using TLSv1.1 or later.

Although it is most widely used in applications to secure web traffic, TLS and SSL are actually general protocols suitable for securing a wide variety of other network traffic that is based on TCP, such as FTP and Telnet.

TLS and SSL provide server authentication, encryption and message integrity. It optionally also supports client authentication.

This document assumes that users who want to use this facility have some basic knowledge of public key cryptography.

For more information on the implementation of TLS and SSL with UniData and UniVerse, refer to *Developing UniBasic Applications* manual for UniData and the *Guide to UniVerse Basic* for UniVerse.

Software requirements

You must have the following applications installed and configured on the client machine.

- Microsoft .NET Framework 4.5
- Microsoft Visual Studio 2012 (required for TLSv1.1 and TLSv1.2)
- UniObjects for .NET 3.171.1 or later

Configuring the database server for TLS/SSL

First, you need to create a Server Security Context Record (SCR).

An SCR contains all SSL related properties necessary for the server to establish a secured connection with a TLS/SSL client. The properties include the server's private key, certificate, client authentication flag and strength, and trusted entities. For more information, see *UniBasic Extensions*.

The SCR can be generated by directly calling the UniData or UniVerse Security API from a BASIC program, or alternatively, by invoking U2 Extensible Admin. The SCR is encrypted by a password and saved in a UniData or UniVerse security file with a unique ID. The path, password and ID of the SCR for a UOJ server are important in the following descriptions.

In order to enable TLS/SSL for UniObjects for .NET on the database server, you need to bind an SCR to the U2 service that runs on the server. You do this by editing the .unisecurity configuration file in the common unishared directory. On UNIX systems, you can determine the location of the unishared

directory by entering "`cat /.unishared.`" On Windows platforms, the default location can be found by examining the following registry record:

`HKEY_LOCAL_MACHINE\SOFTWARE\ROCKET SOFTWARE\unishared`

OR

`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ROCKET SOFTWARE\unishared`

For each SSL-enabled service there should be one line in the .unisecurity file, which has the following format:

`<service_name> <scr_database_path> <scr_ID> <scr_password>`

For UniObjects for .NET, the service name is uvcs, for example:

`uvcs c:\myscrdb myscr "a long passphrase"`

You can share the same SCR for different services, or use a different SCR for each service.

Note: The `<service_name>` must match the server as defined in `unishared/unirpc/unirpcservices`.

UniObjects for .NET secure methods

The following four methods have been added to the UniObjects object for creating a secure connection:

public static UniSession OpenSecureSession (string hostname, string userid, string password, string account, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`X509Certicate clientcertificate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null. The X509Certificate is a Microsoft .NET framework class. For information about this class, see http://msdn.microsoft.com/enus/library/system.security.cryptography.x509certificates.x509certificate.aspx

public static UniSession OpenSecureSession (string hostname, string userid, string password, string account, string service, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`string service` is the type of UniData or UniVerse database account: `udvs` for UniData or `uvcs` for UniVerse.

`X509Certicate clientcertificate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null. The X509Certificate is a Microsoft .NET framework class. For information about this class, see http://msdn.microsoft.com/enus/library/system.security.cryptography.x509certificates.x509certificate.aspx

public static UniSession OpenSecureSession (string hostname, int port, string userid, string password, string account, X509Certificate clientcertificate)

This method returns a new UniSession object, which opens a connection to the UniData or UniVerse database.

`string hostname` is the name or network address of the instance of the UniData or UniVerse database to which to connect.

`int port` is the port number on the host to use for the connection.

`string userid` is the user's login name on the UniData or UniVerse database.

`string password` is the user's password on the UniData or UniVerse database.

`string account` is the name of the UniData or UniVerse database account.

`X509Certicate clientcertificate` is the client certificate in case the server requires client authentication. If the server does not require client authentication, set this value to null. The X509Certificate is a Microsoft .NET framework class. For information about this class, see http://msdn.microsoft.com/enus/library/system.security.cryptography.x509certificates.x509certificate.aspx.

If any of this group of methods fail, it throws an Exception.

public static UniObjects.U2SslProtocols {set; get}

This property gets or sets the flag that indicates the security protocol.

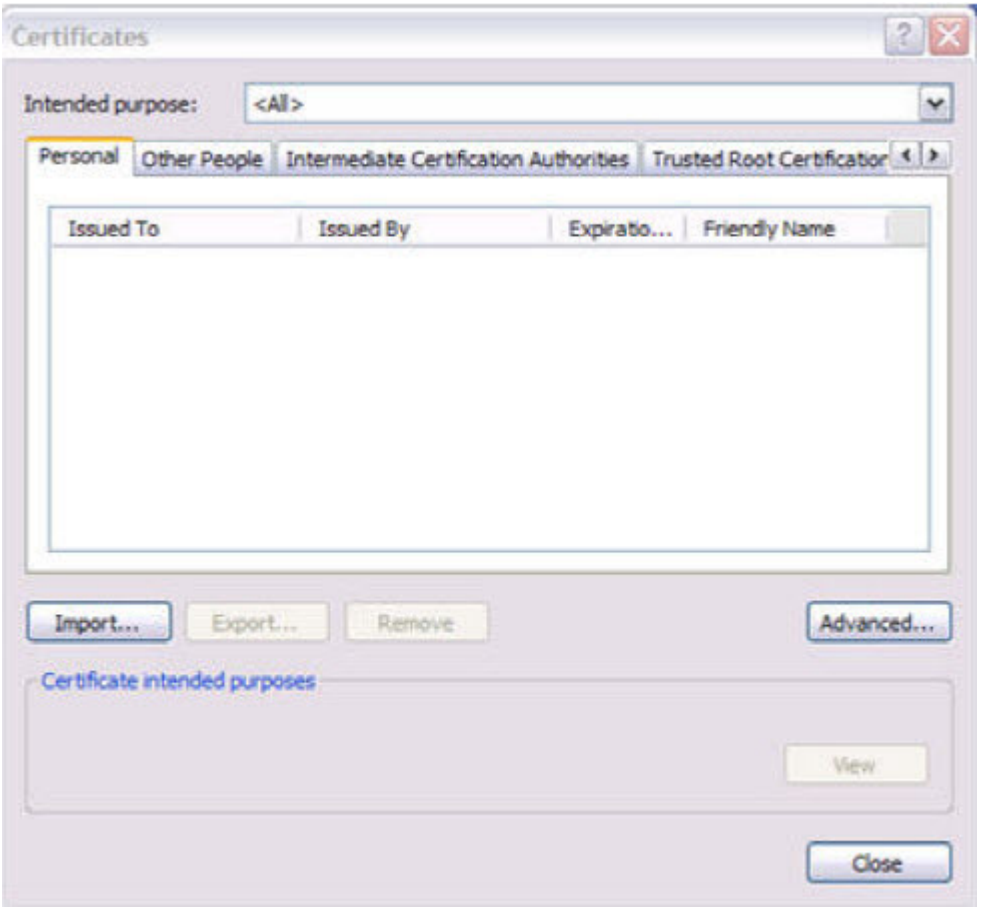
Options are:

- ssl3
- sslv3
- tls1
- tlsv1
- tls11
- tlsv1.1
- tlsv12
- tlsv1.2

Installing a certificate into a Windows certificate store

In order for UniObjects for .NET to function properly, you must install your root certificate in the Windows Trusted Root Certificate Store.

1. Select **Tools > Internet Options** from the Internet Explorer toolbar. The Internet Options dialog box opens
2. Click the **Contents** tab, and then click **Certificates**. A window similar to the following example appears:



3. Click **Import**. Follow the Certificate Import wizard to install your root certificate. The import file name is *your_root_cert_name*, such as myroot.cer. Make sure you import the certificate into the "Trusted Root" certificate store.

Chapter 7: Using code samples

This chapter contains information on using code samples to get up to speed quickly on UniObjects for .NET.

Code samples for UniObjects for .NET

One of the best ways to get a basic understanding of how to write code in the UniObjects for .NET interface is to review code samples for simple software applications.

This section provides information on two code sources:

- Quick guide, on page 165
- Code samples in the installation directory, on page 166

Quick guide

The following code sample written in C# provides a quick guide for software developers working in UniObjects for .NET.

```
using System;
using U2.UODOTNET;
UniSession us1=null;
try
{
        us1=UniObjects.OpenSession("localhost","DENVER\\asmith","xxxi",
        "DEMO","UDCS");

        //open customer file
        UniFile fl=us1.CreateUniFile("customer");

        //use UniDataSet
        string[]sArray={"2","3", "4"};
        UniDataSet uSet=fl.ReadRecords(sArray);
        UniRecord q2=uSet["2"];
        string sq2=q2.ToString();
        UniRecord q3=uSet["3"];
        string sq3=q3.ToString();
        UniRecord q4=uSet["4]'
        string sq4=q4.ToString();

        //use UniCommand
        UniCommand cmd=us1.CreateUniCommand();
        cmd.Command="List VOC SAMPLE 10";
        cmd.Execute();
        string response_str=cmd.Response;
        Console.WriteLine("Response from UniCommand:",response_str);

        //test UniDynArray
        UniDynArray pArray=us1.CreateUniDynArray("a");
        pArray.Insert(1,"b");
        pArray.Insert(1,"c");
        pArray.Insert(1,"d");
        Console.WriteLine("Result from UniDynArray is:
        ",pArray.ToString());
        UniDynArray ur = pArray.Extract(2);
```

```
            Console.WriteLine("Result from UniDynArray Extract is:
            ",ur.ToString());
            ur=pArray.Extract(3);
            Console.WriteLine("Result from UniDynArray Extract is:
            ",ur.ToString());

    }
    catch(Exception ex)
    {
            if(us1 !=null && us1.IsActive)
            {
                UniObjects.CloseSession(us1);
                us1=null;
            }
            MessageBox.Show(ex.Message);
    }
    finally
    {
            if(us1 !=null && us1.IsActive)
            {
                UniObjects.CloseSession(us1);
                us1=null;
            }
    }
```

Code samples in the installation directory

Several more UniObjects for .NET code samples are available in the installation directory, located by default in `C:\U2\UniDK\uonet\samples`. These samples are intended to show software developers how to create software applications, Web applications, and Web services using UniObjects for .NET:

- Creating a simple Windows form application, on page 166
- Creating a simple ASP.NET Web application, on page 166
- Creating an XML Web service, on page 167

Creating a simple Windows form application

The Walkthrough_WindowsAppl project shows the basic steps required to access the UniData database in a simple Windows form application. This sample file is found in the `\Samples\Walkthrough_WindowsAppl` installation directory. The `readme.txt` file contains step-by-step instructions for running and debugging the program.

Creating a simple ASP.NET Web application

The Walkthrough_WebAppl project shows the basic steps required to access the UniData database in a simple ASP.NET Web application. This sample file is found in the `\Samples\Walkthrough_WebAppl` installation directory. The `readme.txt` file contains step-by-step instructions for running and debugging the program.

Before opening and running the sample file, you must complete the following steps to create a virtual folder named Walkthrough_WebAppl in the `\Samples\Walkthrough_WebAppl` installation directory:

1. Run Internet Service Manager (IIS).

2. Right-click **Default Web Site** and select the New/Virtual Directory option.The **Virtual Directory Wizard** starts.
3. Click **Next**.
4. In the **Alias** text box, enter the alias name **Walkthrough_WebAppl**.
5. Click **Next**.
6. Enter the full path of the \Samples\Walkthrough_WebAppl installation directory.
7. Click **Next**.
8. Select the default and click **OK**.

Creating an XML Web service

The Walkthrough_WebService project demonstrates how to create an XML Web service that retrieves data from the UniData database. This sample file is found in the `\Samples\WebService\Walkthrough_WebService` installation directory. The `readme.txt` file contains step-by-step instructions for running and debugging the program.

Before opening and running the sample file, you must complete the following steps to create a virtual folder named Walkthrough_WebService in the `\Samples\Walkthrough_WebService` installation directory:

1. Run Internet Service Manager (IIS).
2. Right-click **Default Web Site** and select the New/Virtual Directory option.The **Virtual Directory Wizard** starts.
3. Click **Next**.
4. In the **Alias** text box, enter the alias name **Walkthrough_WebService**.
5. Click **Next**.
6. Enter the full path of the \Samples\Walkthrough_WebService installation directory.
7. Click **Next**.
8. Select the Default button and click **OK**.

Using an XML Web service

The Walkthrough_Test_WebService project demonstrates how to use an XML Web service that retrieves data from the UniData database. This sample file is found in the `\Samples\WebService\Walkthrough_Test_WebService` installation directory. The `readme.txt` file contains step-by-step instructions for running and debugging the program.

Adding the service description

You can use either of the following methods to specify the location of the service description, which is an XML document that uses the Web Services Description Language (WSDL):

- Add a service description via the command line.
- Add a service description via the Add Web Reference option on the Project menu in Visual Studio .NET.

To add a service description from the command line:

1. At the command line, type the following on one line:

   ```
   wsdl.exe/namespace:UODOTNET http://localhost/
   Walkthrough_WebService/Service1.asmx?wsdl
   ```

   This command creates a C# (Service1.cs) file that contains the proxy code to access the public methods exposed by the XML Web service.
2. Compile this C# file by using the `csc exe` command.
3. Compile the C# program into a DLL that will be used by the ASP.NET application, as follows:

At the command line, type the following on one line:

```
csc /t:library /r:System.Web.Services.dll/out:bin/ u2xmlservice.dll
Service1.cs
```

This command creates a proxy DLL. The application locates the DLL in the bin directory and loads the library automatically.

To add a service description from the Add Web Reference option:

4. Start Visual Studio .NET.
5. On the **Project** menu, click the **Add Web Reference** option.The **Add Web Reference** dialog box appears.
6. In the **URL** text box, type `http://localhost/Walkthrough_WebService/Service1.asmx` and then click the **Go** button to retrieve information about the XML Web service.
7. In the Web reference name box, rename the Web reference to `UODOTNET`, which is the namespace you will use for this Web reference.
8. Click **Add Reference** to add a Web reference for the target XML Web service.Visual Studio .NET downloads the service description and generates a proxy class to interface between this application and the XML Web service.

Chapter 8: Error codes and replace tokens

UniObjects for .NET provides information on replace tokens for error codes and global constants that may be useful in your application. They are contained in the file whose path is `C:\U2\UniClient\UNIDK\INCLUDE\UVOAIF.TXT`. You can add this file to an application through the Add File option of the File menu.

Note: UVOAIF.TXT is a generic file used by client programs accessing the database. This appendix describes only those tokens that are relevant to UniObjects for .NET.

Error codes

These are the error codes that can be returned to a UniObjects application, together with their replace tokens. Each token should be used with the `UniObjectsTokens` prefix—for example, `UniObjectsTokens.UVE_NOERROR`.

Code	Token	Description
0	UVE_NOERROR	No error
14002	UVE_ENOENT	No such file or directory
14005	UVE_EIO	I/O error
14009	UVE_EBADF	Bad file number
14012	UVE_ENOMEM	No memory available
14013	UVE_EACCES	Permission denied
14022	UVE_EINVAL	Invalid argument
14023	UVE_ENFILE	File table overflow
14024	UVE_EMFILE	Too many open files
14028	UVE_ENOSPC	No space left on device
14551	UVE_NETUNREACH	Network is unreachable
22001	UVE_BFN	Bad Field Number
22002	UVE_BTS	Buffer size too small
20003	UVE_IID	Illegal record ID
22004	UVE_LRR	The last record in the select list has been read
22005	UVE_NFI	Not a file identifier
30001	UVE_RNF	Record not found
30002	UVE_LCK	This file or record is locked by another user
30095	UVE_FIFS	The file ID is incorrect for the current session
30097	UVE_SELFAIL	The select operation failed
30098	UVE_LOCKINVALID	The task lock number specified is invalid
30099	UVE_SEQOPENED	The file was opened for sequential access and you have attempted hashed access

Code	Token	Description
30100	UVE_HASHOPENED	The file was opened for hashed access and you have attempted sequential access
30101	UVE_SEEKFAILED	The operation using FileSeek ( ) failed
30103	UVE_INVALIDATKEY	The key used to set or retrieve an @variable is invalid
30105	UVE_UNABLETOLOADSUB	Unable to load the subroutine on the server
30106	UVE_BADNUMARGS	Too few or too many arguments supplied to the subroutine
30107	UVE_SUBERROR	The subroutine failed to complete successfully
30108	UVE_ITYPEFTC	The I-type operation failed to complete correctly
30109	UVE_ITYPEFAILEDTOLOAD	The I-type failed to load
30110	UVE_ITYPENOTCOMPILED	The I-type has not been compiled
30111	UVE_BADITYPE	This is not an I-type, or the I-type is corrupt
30112	UVE_INVALIDFILENAME	Must specify a filename
30113	UVE_WEOFFAILED	WEOFSEQ failed
30114	UVE_EXECUTEISACTIVE	An EXECUTE is currently active on the server
30115	UVE_EXECUTENOTACTIVE	No EXECUTE is currently active on the server
30124	UVE_TX_ACTIVE	Cannot perform this operation while a transaction is active
30125	UVE_CANT_ACCESS_PF	Cannot access part files
30126	UVE_FAIL_TO_CANCEL	Failed to cancel an execute
30127	UVE_INVALID_INFO_KEY	Bad key for the host type
30128	UVE_CREATE_FAILED	The creation of a sequential file failed
30129	UVE_DUPHANDLE_FAILED	Failed to duplicate a pipe handle
31000	UVE_NVR	No VOC record
31001	UVE_NPN	No pathname in VOC record
39101	UVE_NODATA	The server is not responding
39119	UVE_AT_INPUT	The server is waiting for input to a command
39120	UVE_SESSION_NOT_OPEN	The session is not open
39121	UVE_UVEXPIRED	The database license has expired
39122	UVE_CSVERSION	The client and the server are not running at the same release level
39123	UVE_COMMSVERSION	The client or server is not running at the same release level as the communications support
39124	UVE_BADSIG	You are trying to communicate with the wrong client or server
39125	UVE_BADDIR	The directory does not exist or is not a database account

Code	Token	Description
39127	UVE_BAD_UVHOME	Cannot find the UV account directory
39128	UVE_INVALIDPATH	An invalid pathname was found in the UV.ACCOUNT file
39129	UVE_INVALIDACCOUNT	The account name supplied is not an account
39130	UVE_BAD_UVACCOUNT_FILE	The UV.ACCOUNT file could not be found or opened
39131	UVE_FTA_NEW_ACCOUNT	Failed to attach to the specified account
39134	UVE_ULR	The user limit has been reached on the server
39135	UVE_NO_NLS	NLS is not available
39136	UVE_MAP_NOT_FOUND	NLS map not found
39137	UVE_NO_LOCALE	NLS locale support not available
39138	UVE_LOCALE_NOT_FOUND	NLS locale not found
39139	UVE_CATEGORY_NOT_FOUND	NLS locale category not found
39201	UVE_SR_SOCK_CON_FAIL	The server failed to connect to the socket
39210	UVE_SR_SELECT_FAIL	The server failed to select on input channel. When you see this error, you must quit and reopen the session.
39211	UVE_SR_SELECT_TIMEOUT	The select has timed out
40001	UVE_INVALIDFIELD	Pointer error in a sequential file operation
40002	UVE_SESSIONEXISTS	The session is already open
40003	UVE_BADPARAM	An invalid parameter was passed to a subroutine
40004	UVE_BADOBJECT	An incorrect object was passed
40005	UVE_NOMORE	The nextBlock ( ) method was used but there are no more blocks to pass.
40006	UVE_NOTATINPUT	The reply ( ) method can be used only when the response( ) method returns UVS_REPLY
40007	UVE_INVALID_DATAFIELD	The dictionary entry does not have a valid TYPE field
40008	UVE_BAD_DICTIONARY_ ENTRY	The dictionary entry is invalid
40009	UVE_BAD_CONVERSION_ DATA	Unable to convert the data in the field
45000	UVE_FILE_NOT_OPEN	File has been closed, must reopen before performing an operation
45001	UVE_OPENSESSION_ERR	Maximum number of sessions already open
45002	UVE_NONNULL_RECORDID	Cannot perform operation on a nonnull record ID
80011	UVE_BAD_LOGINNAME	The user name or login name provided is incorrect
80019	UVE_BAD_PASSWORD	The password has expired

Code	Token	Description
80144	UVE_ACCOUNT_EXPIRED	The account has expired
80147	UVE_RUN_REMOTE_FAILED	Unable to run as the given user
80148	UVE_UPDATE_USER_FAILED	Unable to update user details
81001	UVE_RPC_BAD_CONNECTION	The connection is bad and may be passing corrupt data.
81002	UVE_RPC_NO_CONNECTION	The connection is broken
81005	UVE_RPC_WRONG_VERSION	The version of the UniRPC on the server is different from the version on the client.
81007	UVE_RPC_NO_MORE_ CONNECTIONS	No more connections available
81009	UVE_RPC_FAILED	The UniRPC failed
81011	UVE_RPC_UNKNOWN_HOST	The host name specified is not valid, or the host is not responding
81014	UVE_RPC_CANT_FIND_ SERVICE	Cannot find the service in the *unirpcservices* file
81015	UVE_RPC_TIMEOUT	The connection has timed out
81016	UVE_RPC_REFUSED	The connection was refused as the UniRPC daemon is not running
81017	UVE_RPC_SOCKET_INIT_ FAILED	Failed to initialize the network interface
81018	UVE_RPC_SERVICE_PAUSED	The UniRPC service has been paused
81019	UVE_RPC_BAD_TRANSPORT	An invalid transport type has been used
81020	UVE_RPC_BAD_PIPE	Invalid pipe handle
81021	UVE_RPC_PIPE_WRITE_ERROR	Error writing to pipe
81022	UVE_RPC_PIPE_READ_ERROR	Error reading from pipe

@Variables

The following tokens represent BASIC @variables:

Value	Token	BASIC @Variable
1	AT_LOGNAME	@LOGNAME
2	AT_PATH	@PATH
3	AT_USERNO	@USERNO
4	AT_WHO	@WHO
5	AT_TRANSACTION	@TRANSACTION
6	AT_DATA_PENDING	@DATA.PENDING
7	AT_USER_RETURN_CODE	@USER.RETURN.CODE
8	AT_SYSTEM_RETURN_CODE	@SYSTEM.RETURN.CODE
9	AT_NULL_STR	@NULL.STR
10	AT_SCHEMA (UniVerse only)	@SCHEMA

Blocking strategy values

The following tokens set the blocking strategy:

Value	Token	Meaning
1	UVT_WAIT_LOCKED	If the record is locked, wait until it is released.
2	UVT_RETURN_LOCKED	Return a value to indicate the state of the lock. This is the default. The values that can be returned are shown in Lock status values.

Command status values

The following tokens represent possible database command status values:

Value	Token	Meaning
0	UVS_COMPLETE	Execution of the command is complete.
1	UVS_REPLY	The command is waiting for a reply.
2	UVS_MORE	More output to come from the command; the command is waiting for a `nextBlock` ( ) method.

Host type values

The following tokens represent possible host type values:

Value	Token	Meaning
0	UVT_NONE	The host cannot be determined or is not yet connected.
1	UVT_UNIX	The host is a UNIX system.
2	UVT_NT	The host is a Windows NT system.

Lock status values

The following tokens represent the values returned by the status ( ) method to indicate the state of a lock:

Value	Token	Meaning
0	LOCK_NO_LOCK	The record is not locked.
1	LOCK_MY_READL	This user holds the READL lock.
2	LOCK_MY_READU	This user holds the READU lock.

Value	Token	Meaning
3	LOCK_MY_FILELOCK	This user holds an exclusive file lock.
4	*no token*	This user holds a shared file lock.
–1	LOCK_OTHER_READL	Another user holds the READL lock.
–2	LOCK_OTHER_READU	Another user holds the READU lock.
–3	LOCK_OTHER_FILELOCK	Another user holds an exclusive file lock.
–4	*no token*	Another user holds a shared file lock.
PID	*no token*	Another user holds a shared file lock. The status value will be the process ID (PID) of the user holding the lock.

Locking strategy values

The following tokens set the locking strategy:

Value	Token	Meaning
0	UVT_NO_LOCKS	No locking. This is the default.
1	UVT_EXCLUSIVE_READ	Sets a READU lock.
2	UVT_SHARED_READ	Sets a READL lock.

FileSeek ( ) pointer values

The following tokens indicate the relative position parameter values used with the `FileSeek` ( ) method of the `UniSequentialFile` object:

Value	Token	Meaning
0	UVT_START	Start of file
1	UVT_CURR	Current position
2	UVT_END	End of file

NLS locale values (UniVerse only)

NLS locale values apply only to UniVerse systems. The following tokens represent the five NLS locale categories:

Value	Token	Category
1	UVT_NLS_TIME	Time
2	UVT_NLS_NUMERIC	Numeric
3	UVT_NLS_MONETARY	Monetary
4	UVT_NLS_CTYPE	Ctype
5	UVT_NLS_COLLATE	Collate

Release strategy values

The following tokens set the release strategy:

Value	Token	Meaning
1	UVT_WRITE_RELEASE	Releases the lock when the record is written.
2	UVT_READ_RELEASE	Releases the lock when the record is read.
4	UVT_EXPLICIT_RELEASE	Maintains locks as specified by the lock strategy. Releases the locks only with the `unlockRecord()` method.
8	UVT_CHANGE_RELEASE	Releases the lock when a new record ID is set. This value is additive and can be combined with any of the other values.

System delimiters

The following tokens represent database system delimiters:

Value	Token	Character value	Meaning
1	TM_CHAR	251	Text mark
2	SM_CHAR	252	Subvalue mark
3	VM_CHAR	253	Value mark
4	FM_CHAR	254	Field mark
5	IM_CHAR	255	Item mark

Encryption values

The following tokens set the encryption values:

Value	Token	Meaning
0	NO_ENCRYPTION	Do not encrypt data.
1	UV_ENCRYPT	Encrypt data using internal database encryption.