Connecting to Kdb+ from C# Platforms

AquaQ Analytics Limited

Authors

This document was prepared by:

Danny O' Hanlon Paul McCabe Chris Patton Ronan Pairceir Dermot French Kent Lee Damien Colton		

Revision History

Version Number	Revision Date dd/mm/yyyy	Summary of Changes	Document Author
1.0	01/01/2013	Initial Release	See list above

Table of Contents

1 Company Background 4

2 Introduction 5

3 Kdb Connections App 6

4 Setting Up a q session 8

5 Connecting to the local q session 9

6 The c.Flip object 11

6.1 Converting a List into a c.Flip 13

7 Loading Student List into Kdb+ 14

8 Simple Querying 16

8.1 Accessing the data in a c.Flip Object 16

8.2 Determining the number of returned rows 18

9 Mapping Data 20

10 Executing Parameterised Queries 23

10.1 Single Name 23

10.2 Age Range 24

10.3 Query With List Of Names 24

10.4 Query List of Names and an Age 25

10.5 Query List of Names, List of Ages and Gender 26

11 Casting Simple Data Types 28

12 Error Handling 29

1 Company Background

AquaQ Analytics Limited (www.aquaq.co.uk) is a provider of specialist data management, data analytics and data mining services to clients operating within the capital markets and financial services sectors. Based in Belfast, the company was set up in April 2011. Our domain knowledge, combined with advanced analytical techniques and expertise in best-of-breed technologies, helps our clients get most out of their data.

The company is currently focussed on three key areas:

- Kdb+ Consulting Services – Development, Training and Support (both onsite and offsite).
- Real Time GUI Development Services (both onsite and offsite).
- SAS Analytics Services (both onsite and offsite).

For more information on the company, please email us on info@aquaq.co.uk.

2 Introduction

In order to connect to and interact with a kdb+ database from C#, an interface class, *c.cs*, is required (available from kx systems at http://kx.com/q/c/). This class contains the methods and objects needed to interact between the two languages. We have found in working with *c.cs* that there is a lack of accompanying detailed documentation and sample code. This makes it difficult to decipher and use this class, unless the developer has previous experience of it.

This document aims to provide some guidance for users wishing to use *c.cs* to communicate between C# and kdb+ to accomplish some of the following tasks:

- connecting to the kdb+ database from C#,
- loading data,
- executing queries, and
- handling the data that is returned from these queries in C#.

The chapters below provide explanations of the code, worked examples which use simple tables and subsequently build up to more difficult queries. Error handling is also discussed. The document assumes that the reader is familiar with .NET but has limited kdb+ and q knowledge.

3 Kdb Connections App

This document should be used together with the *KdbConnections* project code, which is a Windows Presentation Foundation (WPF) application written in C#. The availability of this sample application code, along with this document, allows the actual code to be read, played with and debugged to aid understanding. The sample application allows a user to connect to a q session, load data into the database, query the database and handle the returned data.

The sample data used in the project is student data. The examples involve querying a student table in a kdb+ database, a collection of data listing name, age, gender, date of birth and outstanding course fee for each student. This data is initially contained within the Application as a List which can be viewed in a grid, and then loaded into the database.

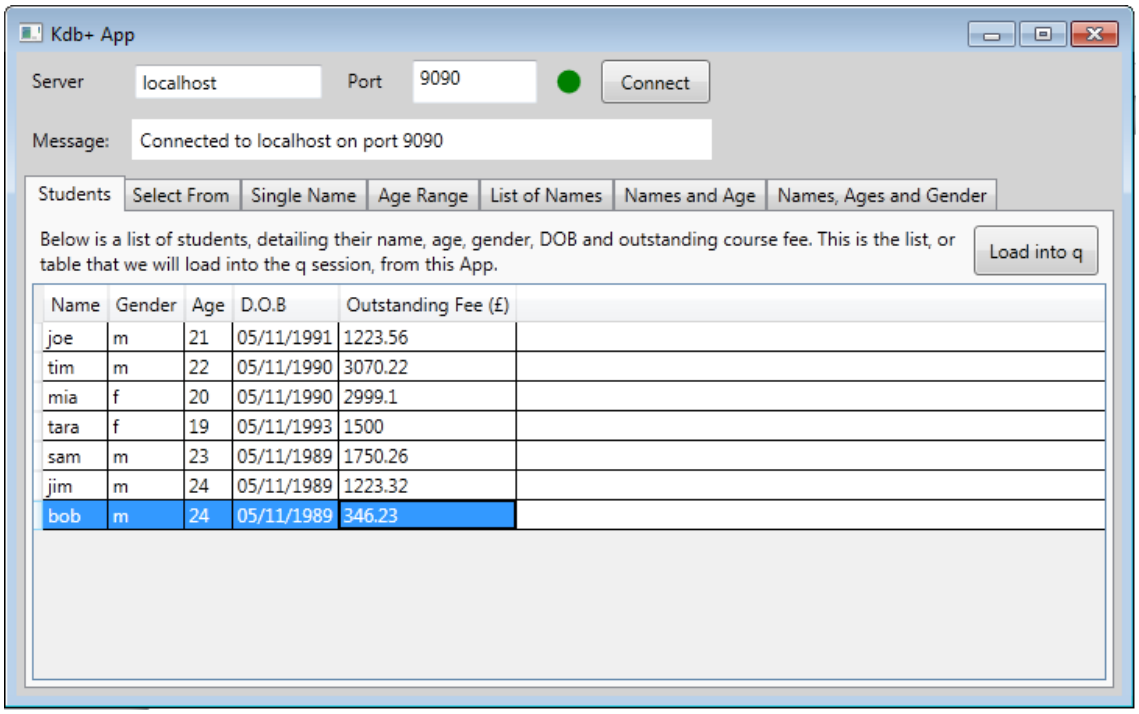

Figure 1 The Kdb Connections App showing the initial student data to be loaded into the database

The architecture of the application itself does not follow any particular model; instead it is written in such a way as to allow easy interpretation of the code.

The project has five main classes:

- *Student,*
- *DBConnections.cs,*
- *Queries.cs,*
- *c.cs, and*
- *MainWindow.xaml.cs.*

The partial c.cs class is Kx's API for C# and should be included in the project. This class is inherited from *TcpClient*. It also contains other classes such as *Flip* and *Dict* that will be used in handling data. Note the namespace of the class is *kx*. This will need to be included in any references (using *kx;*) where the c methods or objects are being used. It is suggested that connections and querying methods are separated out from other functional code to minimise coupling of classes.

The static *DBConnection* class contains the connection property itself and the connect method. This requires the *c.cs* class. The connection object is an instantiation of the c class, and all querying is facilitated through this property.

The *Queries* class contains all of the methods required for querying the database and loading data. The data passed through this class is generally in the form of a generic object x, or specific kx objects such as *c.Flip*. This also helps to reduce coupling of classes with project specific objects such as *Student*.

4 Setting Up a q session

The first step is to download and install kdb+ onto a server or a local PC. A trial version of kdb+ is available from http://kx.com/trialsoftware.php).

Once kdb+ is installed, a q session should be started (refer to trialsoftware documentation for details of how to start a q session). The q session can be running on a remote server if available, or locally on a host PC (localhost). Once a q session has been started, a port must be specified within the session to allow a connection to be opened for communication between the q session and the C# code (note that both the server name and the port number are required to communicate to the q service). The port can be set by entering the following in the command line of the q session (note that any suitable open port can be specified other than 9090 below):

\p 9090

If successful, a new command line will appear q>. Now there is a q session running that a connection can be made to from C#.

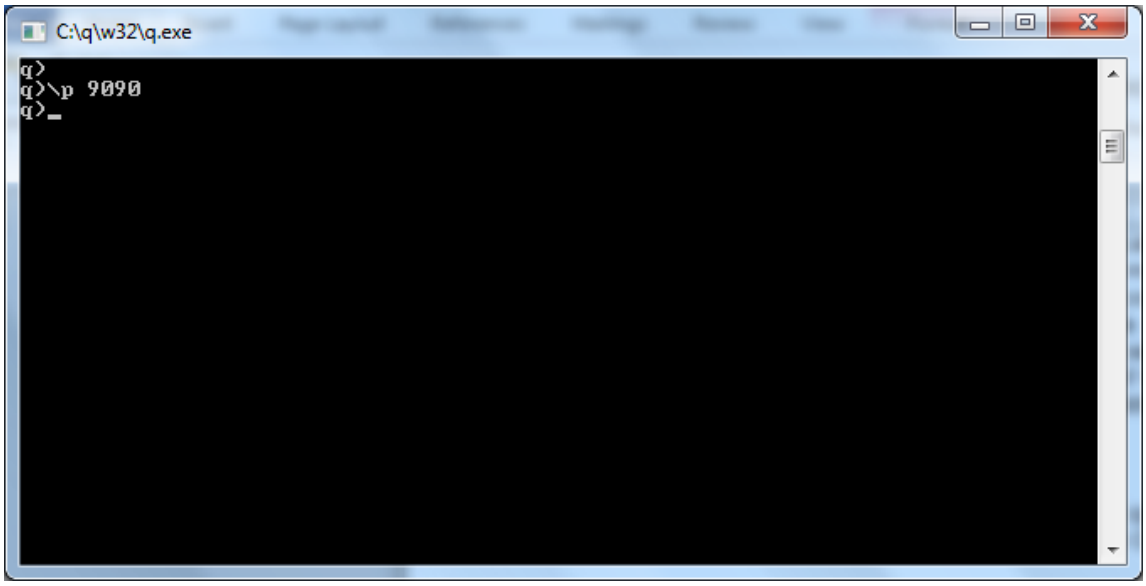

Figure 2. setting the port in the q session

The next step is to connect to this q service from C#.

5 Connecting to the local q session

The *connect* method has been separated out into the static *DBConnection* class. This class also contains a property, *Connection*. This is an instance of the c class, and as such, this serves as the object through which querying and passing of data will occur.

```
using kx;

namespace KdbConnections
{
    public static class DBConnection
    {
        private static c _connection;
        public static c Connection
        {
            get { return _connection; }
            set { _connection = value; }
        }

        public static bool ConnectToKDB(string host, int port)
        {
            bool connected = false;

            _connection = new c(host, port);

            if (_connection.Connected)
            {
                connected = true;
            }

            return connected;
        }
    }
}
```

The *ConnectToKDB* method connects to the database by creating a new instance of the c class, which is the Connection property. The class constructor has three overloads (Fig 3). The simplest takes a string and int, representing the *server* and *port number* for the q service we wish to connect to. The second overload accepts these two arguments plus a second string, which is the *username*. The third overload accepts these three arguments plus a further int, which is the *Max buffer size* of returned data.

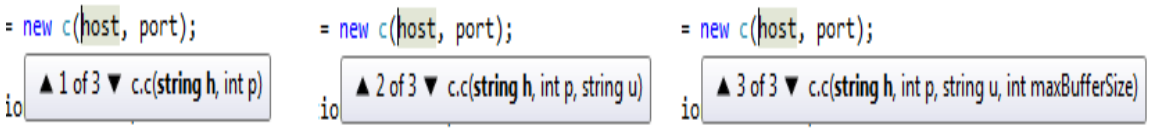

Figure 3 Creating a new instance of the c class passing in a host and a port number sets up a connection

To create the connection for the application, the first overload is used. The host and port are passed into the *ConnectToKDB* method from the *MainWindow.xaml.cs* class, which reads the host and port from the textboxes in the application.

When a new instance of this c class is created, the connection is made to the q session. This instance, now known as the Connection property has a property, *Connected*, which returns a bool indicating the connection status. Having instantiated the connection, the *ConnectToKDB* method then checks the value of *Connection.Connected*. It returns this to the code behind the class, allowing it to be presented to the user in the form of the connection status indicator in the application.

Assuming a successful connection has been made to the q session, data can now be loaded into the database from the C# application and queries can then be run against this data. Consider the list of Students displayed in the application. This data will form the table in the database that the example queries will be ran against. However, the first task is to load this data into the q session. In order to pass data into the database (and equally when data is returned from a query run against the database), it is packaged as a *c.Flip* object, discussed in Chapter 3.

6 The c.Flip object

The *c.Flip* object holds the data in a structured format that q can decipher. Information can be passed into the database as a *c.Flip*, and when queries are made, the returned data is cast as a *c.Flip* object before the information is extracted from it.

The *c.Flip* contains two properties, x and y. The x property of the flip object is always a string array, containing the column headers. As such, the number of elements in the array is always equal to the number of columns in the table. Consider the list of Students in the application. This data needs to be contained in a flip object to be loaded to the database. The following shows how this data is arranged in the flip, before detailing how to package it into this object.

Name	Gender	Age	D.O.B	Outstanding Fee (£)
joe	m	21	05/11/1991	1223.56
tim	m	22	05/11/1990	3070.22
mia	f	20	05/11/1990	2999.1
tara	f	19	05/11/1993	1500
sam	m	23	05/11/1989	1750.26
jim	m	24	05/11/1989	1223.32
bob	m	24	05/11/1989	346.23

Figure 2 The student table in the Kdb Connections App

Figure 5 shows the x property of the *c.Flip* containing the student data. The five column names are held in the string array.

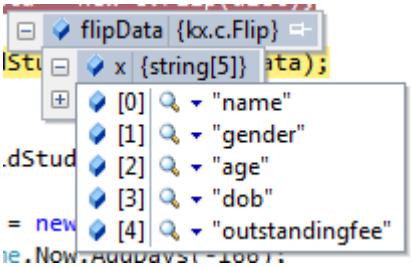

Figure 3 The x property of a c.Flip is a string array holding the column names

The y property (Fig 6) is more complicated in that it is an object array. Within y, each element is another array which contains the data from each column in the table. Each of these contained arrays is strongly typed.

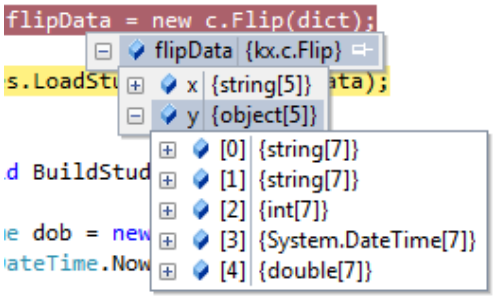

Figure 4 The y property of a c.Flip is an object array, each element being a typed array containing the data from the columns in the table

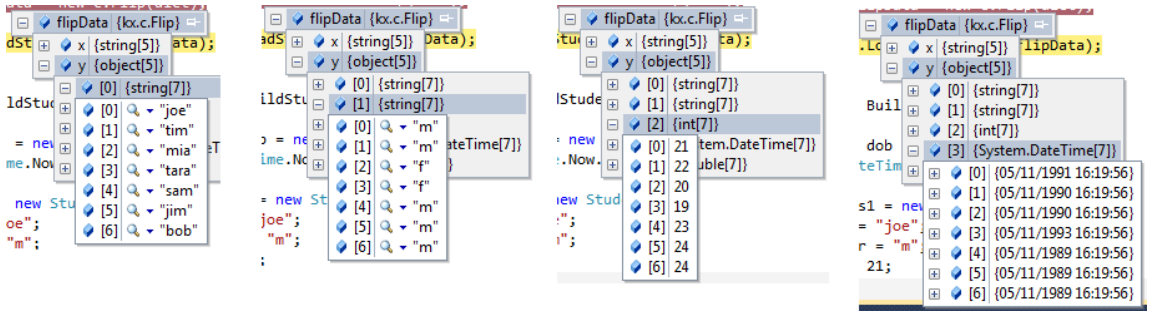

Figure 5 The arrays contained within the y property, showing how they contain the column data in typed arrays

The above shows the column data for four of the five arrays. The first array in the y property, y[0], contains the names, y[1] contains the genders, y[2] contains the age data and y[3] contains the date of births.

Consider the information regarding student Joe. Joe is the first student in the table, and all the information relating to him is contained in the first row of the table. In the *c.Flip* object, all the data relating to Joe is contained across the five arrays of the y-property, at the same element index in each array, in this case [0].

As a guide to visualising how the data is organised in the c.Flip, compare the data in the table below with how it is organised in the arrays. Tim's gender would be located in the second array in the y property, in element 1

"name",	"gender",	"age"
joe,	m,	21
tim,	m,	22
mia,	f,	20

[0,0], [1,0], [2,0]
[0,1], [1,1], [2,1]

6.1 Converting a List into a c.Flip

The *ConvertListIntoFlip* method included below shows the code to convert the List of Students into the *c.Flip* object required to load it into the database.

```
private void ConvertListIntoFlip()
{
       string[] colNames = new string[5];
       colNames[0] = "name";
       colNames[1] = "gender";
       colNames[2] = "age";
       colNames[3] = "dob";
       colNames[4] = "outstandingfee";

       object[] colData = new object[5];

       string[] names = new string[StudentNames.Count];
       string[] genders = new string[StudentNames.Count];
       int[] ages = new int[StudentNames.Count];
       DateTime[] dobs = new DateTime[StudentNames.Count];
       double[] outstandingFees = new double[StudentNames.Count];

       int a = 0;
       foreach (Student stud in StudentNames)
       {
            names[a] = stud.Name;
            genders[a] = stud.Gender;
            ages[a] = stud.Age;
            dobs[a] = stud.DOB;
            outstandingFees[a] = stud.OutStandingFee;

             a++;
       }

       colData[0] = names;
       colData[1] = genders;
       colData[2] = ages;
       colData[3] = dobs;
       colData[4] = outstandingFees;

       c.Dict dict = new c.Dict(colNames, colData);
       c.Flip flipData = new c.Flip(dict);

       _queries.LoadStudentsToDB(flipData);
}
```

The x property will be the string array, *colNames*. This is populated by setting each element to whatever the column titles should be, and the size of the array should equal the number of columns in the table.

The data from the student properties is then stored in strongly typed arrays, one for each student property. The size of these arrays is equal to the

number of students in the list being loaded into the database. Once these arrays have been defined, it is a case of using a foreach loop against the Student list, together with an increasing int value to loop through and add each student's data to the arrays. The int is used to define which element to store the data in in each array. For one student, the int will be constant and thus that student's data will be stored in the same 'row.' These typed arrays are then held inside an object array, *colData*, which forms the y property of the *c.Flip*.

These arrays cannot be assigned directly to a new *c.Flip* object. Instead, to create a new *c.Flip* object requires a *c.Dict* object to be passed into the *c.Flip* class constructor. The *c.Dict* is created by instantiating a new *c.Dict* class, the constructor taking two argurments, both generic objects. In this case, the string array *colNames* is passed in along with the object array *colData*. This dict object is then passed into the *c.Flip* class constructor, allowing a new flip object to be created, which contains the student data.

7 Loading Student List into Kdb+

The student table can then be loaded into the database using this *c.Flip*. This is achieved using the *c.k* method. The *c.k* method has six overloads, which are combinations of accepting strings and objects as arguments. The method returns a generic object, which when use to query, will be the data from that executed query.

The Flip is passed into the *LoadStudentsToDB* method below. Null and connection status checks are performed, and if passed, the method executes the line that will load the flip object into the database as a new table. The *c.k* method used below accepts three arguments, two strings and the *c.Flip* object. The table will be named 'student' on the database, indicated by the second string passed in.

```
public void LoadStudentsToDB(c.Flip flipData)
{
	if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
	{
		DBConnection.Connection.k("{[x;y]x set y}","student", flipData);
```

To check the table has been loaded in q, type *student* into the command window followed by enter. If the data is loaded, the table should be displayed.

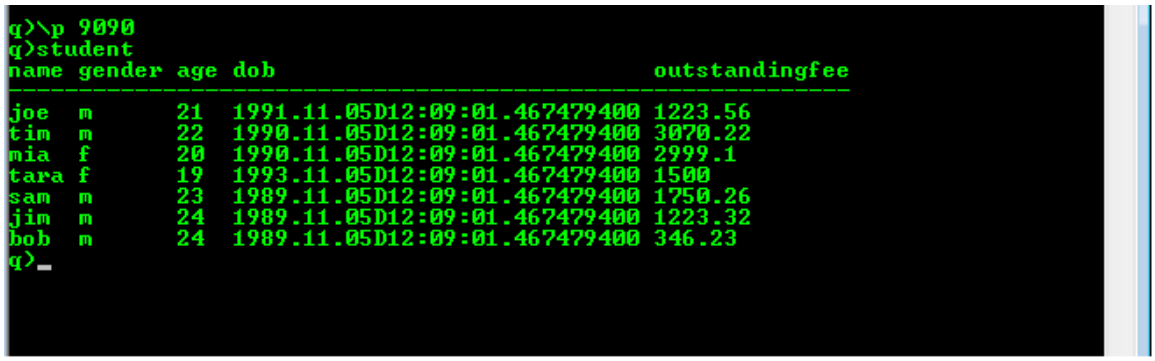

Figure 6 Once loaded from the application, the student table can be displayed in the q session by typing *student* followed by enter

8 Simple Querying

With the data now loaded into the database, this table can be queried. Query statements can be submitted to q by passing them in as a string via the *c.k* method. The object that is returned from the *.k* method contains the data from the executed query.

The method below, *SelectFromStudentQuery*, shows a simple query being executed which will return back all the data in the student table. As mentioned, the k method has six overloads but the simplest accepts just a string as an argument, which in this case is the query statement *select from student*

```
public c.Flip SelectFromStudentQuery()
{
	if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
	{
		object obj = DBConnection.Connection.k("select from student");
		return c.td(obj);
	}
	else
	{
		return null;
	}
`
```

The returned object needs to be converted into a *c.Flip* in order to make the data accessible and allow it to be presented. This is achieved using the *c.td(object x)* method as follows:

```
c.Flip flipData = c.td(obj);
```

It is this *c.Flip* object that the returned data is extracted from. It is returned from the *SelectFromStudentQuery* method to the code behind, where the data is pulled out and mapped to the relevant fields.

8.1 Accessing the data in a c.Flip Object

To access or extract a particular piece of data in the *c.Flip* object requires using the *c.at* method. This method returns an object, having passed in an object x, and int i. The object passed in is a specific array in the *c.Flip*, and the int defines which element to target in that array.

```
public static object at(object x, int i)
{
       object r = ((Array)x).GetValue(i);
       return qn(r) ? null : r;
}
```

Consider the first student listed in the table, Joe, m, 21. This data relating to Joe is contained in the five arrays of the y property, specifically in the first element, index 0, of each array. The name is contained in the first column of data, which is array y[0]. The gender and age information are contained in the y[1] and y[2] arrays respectively.

To access the name data, the specific 'co-ordinates' of Joe's name data is passed into the .at method.. The name data is contained in y[0], and for Joe is the first row or element, [0]. To access Joe's name from the flip object would require passing the following into the .at method.

```
(flipData.y[0], 0)
```

The same logic applies to the gender and age data, contained in the arrays y[1] and y[2] so for the gender and age data respectively would require:

```
(flipData.y[1], 0)
(flipData.y[2], 0)
```

Using this information, a new student object can be created and populate its properties with Joe's data.

```
Student student = new Student();
student.Name = (c.at(flipData.y[0], 0).ToString());
student.Gender = (c.at(flipData.y[1], 0).ToString());
student.Age = Convert.ToInt32(c.at(flipData.y[2], 0));
student.Age = Convert.ToInt32(c.at(flipData.y[2], 0));
student.DOB = Convert.ToDateTime(c.at(flipData.y[3], 0));
student.OutStandingFee = Convert.ToDouble(c.at(flipData.y[4], 0));
```

As the *.at* method returns an object, it needs to be converted to a type to be populate the Student object properties. Name and Gender are both strings, stored as syms in the q table, so the returned data is converted *ToString()*. The age is an int32. The result is a Student object, Name = joe, Gender = m and Age = 21.

In order to create student objects for the others included in the student table, the integer passed into the *.at* method needs to be increased, to move onto the next element in each array. This can be done manually, by copying

the method above for each student required and increasing the int by 1 each time, but this involves a lot of repetition of code.

A neater approach is instead to pass in the int relating to the element index, as a variable. The variable should increase automatically each time using the *int*++ method, within a specified range of values. This range of values starts at 0, and continues until the last row in the table, which is 7 for the student table. Realistically, the number of rows in the data being returned will be unknown and thus the code needs to be able to handle this.

8.2 Determining the number of returned rows

The c class contains a method *n* that can be used to determine the number of rows or the number of columns in the returned data.

```
public static int n(object x)
{
            return x is Dict ? n(((Dict)x).x) : x is Flip ? n(((Flip)x).y[0]) :
x is char[] ? e.GetBytes((char[])x).Length : ((Array)x).Length;
```

The *c.n* method accepts an object; in this case that object will be either the x property of the *c.Flip*, or a specific element in the y property, which will be a typed array.

By passing in the x property of the *c.Flip*, the int returned from the *n* method is the number of elements in x. This is equal to the number of columns of data. Similarly, passing in any array in the y property, returns the number of elements in that array. This is akin to the number of rows of returned data.

```
int cols = c.n(data.x);
int rows = c.n(flipData.y[0]);
```

In the case of the student table, cols should equal 5 (name, gender, age, dob and outstanding fee). It's acceptable to assert the number of elements contained in any one of the y property arrays, as they all contain the same number of elements.

With this information, a *for* statement can be used to repeat the previous method of extracting the returned data to a Student object and adding them to a list; as follows:

```
List<Student> students = new List<Student>();

for (int a = 0; a < rows; a++)
{
        Student student = new Student();
        student.Name = (c.at(flipData.y[0], 0).ToString());
        student.Gender = (c.at(flipData.y[1], 0).ToString());
        student.Age = Convert.ToInt32(c.at(flipData.y[2], 0));
        student.Age = Convert.ToInt32(c.at(flipData.y[2], 0));
        student.DOB = Convert.ToDateTime(c.at(flipData.y[3], 0));
        student.OutStandingFee = Convert.ToDouble(c.at(flipData.y[4], 0));

        students.Add(student);
```

Instead of specifying the row number as done previously it is assigned the variable *a*, which will increment by 1 on each iteration, until *a* is equal to the number of elements. Each student is added to `List<Student> students` at the end of each iteration. Now the method is more concise and will create only as many student objects to add to the list as is required. However, there is a major drawback in using this method - relying on the columns in the table being in a particular order; Name, Gender, Age, DOB and Fee, as it specifies which array in y to get the correct data from. For the simple student table, this is fine. But for much larger databases, in which columns may be removed, added or re-ordered the method above breaks down and needs to be re-coded to the new table layout.

An alternative approach is to use a *Map* method that employs a *switch* statement, interrogating the column name and then assigning it to the relevant property.

9 Mapping Data

A *MapColumnData* method has been set up in the *Student* class. This is a more reliable way of assigning data from the *c.Flip* object to the relevant properties in an object. The method requires three arguments to be passed into it; a piece of data(type object) that is to be assigned to a Student property, the column name (string) that the data was contained in, and an instance of the *Student* class itself. The method returns a Student.

The *MapReturnedData* method below deals with obtaining the correct information in preparation for passing it into the *MapColumnData* method. *ReturnedStudentData* is a List<Student> that the students will be added to. The *c.Flip* object(flipData) contains the data returned from a query.

```
private void MapReturnedData(c.Flip flipData)
{
       ReturnedStudentData.Clear();

       int rows = _queries.GetNumberOfRows(flipData);
       string[] colNames = _queries.GetColumnNames(flipData);

       for (int a = 0; a < rows; a++)
       {
                Student student = new Student();

                for (int col = 0; col < colNames.Length; col++)
                {
                    string colname = colNames[col];
                    object data = c.at(flipData.y[col], a);

                    student.MapColumnData(colname, data, student);
                }

                ReturnedStudentData.Add(student);
```

An int (rows) is obtained using the *c.n* method to assert the number of rows (number of elements in any of the y property arrays) in the *flipData*. A string[] (colNames) is obtained by taking a copy of the *c.Flip.x* property, as this is already a string[]of the table column names. These two objects are obtained via the *Queries* class (_queries) and their methods are included in an insert below.

With these objects now obtained, an initial *for* loop is used that will iterate for each row of data returned. For each iteration, or row of data, a new Student object is created. Then, for each of the columns within this row of data, the column name together with the particular piece of data obtained

and the student object, are passed into the *MapColumnData* method in the Student class.

```
public string[] GetColumnNames(c.Flip flipData)
{
       string[] colNames = new string[0];

       if (flipData != null)
       {
                colNames = flipData.x;
       }

            return colNames;
}
```

```
public int GetNumberOfRows(c.Flip flipData)
{
       if (flipData != null)
       {
              return c.n(flipData.y[0]);
       }
       else
       {
                return 0;
       }
}
```

```
public Student MapColumnData(string colName, object data, Student student)
{
      if (data != null)
      {
               switch (colName)
               {
                   case "name": //string
                       student.Name = (string)data;
                       break;
                   case "age": //int
                       student.Age = (int)data;
                       break;
                   case "gender": //string
                       student.Gender = (string)data;
                       break;
                   case "dob": //DateTime
                       student.DOB = (DateTime)data;
                       break;
                   case "outstandingfee": //double
                       student.OutStandingFee = (double)data;
                       break;
                   default:
                       break;
               }
      }
```

```
        return student;
}
```

After a null check around the data passed in, a *switch* statement is used on the column name passed it. It looks to match the *colName* to hardcoded column names that it is expected. For example, the data passed it has come from colName="name". When this is matched with a case, the method takes that data and assigns it to the property within that case statement.

As the data is of type object, it needs to be cast to the correct type as the property, and this is done here via the (string)data, int(data), double(data) or DateTime(data) in each case statement. Each case statement must end with a break; indicating that the case has been met and executed, and there is no need to continue through any remaining cases.

The switch statement must always end with default:break; indicating the end of the statement. The student object, with the data now assigned to a property on the object, is returned.

The iteration in the *MapReturnedData* method then repeats for a different column name, mapping this to another property, until all the return columns data have been mapped, and then it moves onto the next row of data.

This method is much more robust than the previous approach of specifying which array in the y-property to take the data from. Using the *MapColumnData* method, if columns in the table are re-arranged, the method still functions correctly as it looks to match the column name with what is expected before assigning it to that property. The method only needs to be altered if the column names in the database are renamed, so that the `case` statements match the column names. If new columns are added or removed, new case statements can be added in any order.

10 Executing Parameterised Queries

The first query example involved submitting the query as a string using the k method. This k method has an overload accepting an object x, so rather than passing in a query string, an object array can be passed in containing multiple parameters to query with and q can interpret the object array, provided it is in the correct format, and query accordingly. Consider now passing object[] *queryParams* to *c.k*.

queryParams, must be defined with a size and have each element populated. The first element of is always the query statement itself, and must be in the form of a character array. The query statement itself changes format from previous examples. The follow examples deal with building up this object array for various scenarios in querying the example student table.

10.1 Single Name

Consider the Single Name tab in the application. Selecting a name in the dropdown and querying for this name is carried out via the following:

```
public c.Flip QueryWithSingleNameAsParam(string selectedName)
{
       object[] queryParams = new object[2];

       string query = "{[selectedname]select from student where
name=selectedname}";
       queryParams[0] = query.ToCharArray();
       queryParams[1] = selectedName;

       if (DBConnection.Connection != null &&
       DBConnection.Connection.Connected)
       {
              object obj = DBConnection.Connection.k(queryParams);
              return c.td(obj);
       }
```

The object array, *queryParams*, is defined as containing two elements. The first element is always the query statement, in this case "{[selectedname]select from student where name=selectedname}". The query statement needs to be in the form of a character array, as such it is converted using *ToCharArray* and added as the first element of *queryParams*. The second element *queryParams* is a string, the selected name that we wish to query. As it's a object array, the second element of *queryParams* is set to be *selectedName*.

With both elements populated, *queryParams* is passed into the *.k*, querying the database and returns an object which can be handled in the same way as previously detailed.

10.2 Age Range

The table can be queried for an age range after specifying an upper and lower limit. Having entered the lower limit age, *fromAge*, and the upper limit, *toAge*, pass these into the method below to query to the database.

```
public c.Flip QueryAgeRange(int fromAge, int toAge)
{
       object[] queryParams = new object[3];
       string query = "{[fromAge;toAge]select from student where age >=
fromAge,age <= toAge}";

       char[] queryarray = query.ToCharArray();

       queryParams[0] = queryarray;
       queryParams[1] = fromAge;
       queryParams[2] = toAge;

       if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
       {
              object obj = DBConnection.Connection.k(queryParams);
             return c.td(obj);
        }
       else
       {
                return null;
```

queryParams now consists of three elements; the query statement and two values to query the age range. The two age values to query between are int values, and are added to the *queryParams* as the second and third element. The query statement instructs the database to look for students with an age >= fromAge, AND <= toAge.

10.3 Query With List Of Names

Should it be necessary to query with a list of multiple values, such as a list of names or ages, this can be achieved by converting the List<T> into an array

T[]. Consider a list of student names (List<string> namesToQuery) we wish to query against. First it is converted into a string[] via the following.

```
string[] names = new string[namesToQuery.Count];

for (int a = 0; a < namesToQuery.Count; a++)
{
	names[a] = namesToQuery[a];
```

The string[] names must have a specified size, equal to the number of names in the list. As such, the size is set to equal to the count of the list. Using a *for* loop that counts up to the number of items in the list, each name is added to the string array. This array can then be added to *queryParams* together with the query statement, and passed to the database as below:

```
public c.Flip QueryStudentListOfNames(List<string> namesToQuery)
{
	object[] queryParams = new object[2];

	string query = "{[names]select from student where name in names}";
	char[] queryarray = query.ToCharArray();

	string[] names = new string[namesToQuery.Count];

	for (int a = 0; a < namesToQuery.Count; a++)
	{
		names[a] = namesToQuery[a];
	}

	queryParams[0] = queryarray;
	queryParams[1] = names;

	if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
	{
		object obj = DBConnection.Connection.k(queryParams);
		return c.td(obj);
	}
	else
	{
```

10.4 Query List of Names and an Age

Querying with a list can be combined with other parameters, such as a lower age range. Using techniques from the previous two examples, they can be combined to query the student table against a list of names and a lower age range.

```
public c.Flip QueryStudentListOfNamesAndAge(List<string> namesToQuery, int
fromAge)
{
	object[] queryParams = new object[3];

	string query = "{[names;fromAge]select from student where name in
names,age >= fromAge}";
	char[] queryarray = query.ToCharArray();

	string[] names = new string[namesToQuery.Count];

	for (int a = 0; a < namesToQuery.Count; a++)
	{
		names[a] = namesToQuery[a];
	}

	queryParams[0] = queryarray;
	queryParams[1] = names;
	queryParams[2] = fromAge;

	if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
	{
		object obj = DBConnection.Connection.k(queryParams);
		return c.td(obj);
	}
	else
	{
```

10.5 Query List of Names, List of Ages and Gender

There is no restriction on the combination of parameters that can be used, they can be simple single items or multiple lists and ranges. The previous examples should have provided a good foundation allowing you to tailor queries to particular needs.

This is the most complex example yet, combing the previous methods to query the student table with a list of names, a list of ages and specifying a gender. The *queryParams* array consists four elements; the char[] of the query statement, string[] of names, int[] for ages, and the final element is the gender as a string.

```
public c.Flip QueryNamesAgesAndGender(List<string> namesToQuery, List<int>
agesToQuery, string genderToQuery)
{
	object[] queryParams = new object[4];

	string query = "{[names;ages;genders]select from student where name in
names,age in ages,gender in genders}";

	char[] queryarray = query.ToCharArray();

	string[] names = new string[namesToQuery.Count];
	int[] ages = new int[agesToQuery.Count];

	string[] gender = new string[1];

	for (int a = 0; a < namesToQuery.Count; a++)
	{
		names[a] = namesToQuery[a];
	}

	for (int b = 0; b < agesToQuery.Count; b++)
	{
		ages[b] = agesToQuery[b];
	}

	queryParams[0] = queryarray;
	queryParams[1] = names;
	queryParams[2] = ages;
	queryParams[3] = genderToQuery;

	if (DBConnection.Connection != null &&
DBConnection.Connection.Connected)
	{
	       object obj = DBConnection.Connection.k(queryParams);
		return c.td(obj);
	}
	     else
```

11 Casting Simple Data Types

Kdb+ has data types similar to C#. Table 1 shows the equivalent data types between the two languages.

KDB+	**C#**
boolean	boolean
short	Int16
Int	Int32
long	Int64
float	double
char	char
symbol	string
datetime	DateTime
time	TimeSpan
string	CharArray

Table 1 q data types and their C# equivalents

12 Error Handling

A try-catch can be wrapped around the *Connect* method from Chapter 2, handling any errors in connecting. Consider the *ConnectToKDB* method inset below.

```
private void ConnectToKDB()
{
       try
       {
                bool connected = DBConnection.ConnectToKDB(_host, _port);

                if (connected)
                {
                   // Set relavent display properties indicating to user
                      connection was successful
                }
       }
       catch (SocketException e)
       {
                txtErrorMessage.Text = "Cannot connect to " + _host + " on port "
                + _port + " " + e.Message;
       }
       catch (Exception exc)
       {
                txtErrorMessage.Text = "Cannot connect to " + _host + " on port "
                 + _port + " " + exc.Message;
       }
}
```

Inside the *try*, the *Connect* method in the *DBConnection* class is called to create the connection.

Two *catch* statements are then included, one for a *general Exception* and one for a *Socket Exception*. Should the *DBConnection.ConnectToKDB* method return an error, it will be caught by the relevant exception. The message associated with this exception is then presented to the user in the front end of the application.

A *SocketException* will catch any errors in connecting to the database related to an incorrect server or port number being supplied.