it's about time

Technical Whitepaper

Java API for kdb+

Date May 2018

Author Peter Lyness

Contents

Java API for kdb+

Java as a programming language has been consistently popular for almost two decades across many measurements, highlighting its importance in past and present development environments. This, combined with its longevity and compatibility of code between versions and operating systems, means that the landscape of Java applications in many industries is one very much divided between new offerings and long-established legacy code.

Financial technology is no exception to this. As a domain that is driven by competition to push boundaries whilst paradoxically being risk-averse, there is an inevitable mixture of contemporary and legacy code in many production systems. Because of this, it is of vast importance that developers have access to tools necessary for communication and integration within these systems and that the risk of implementation is minimized to the highest degree possible. Kdb+ as a technology is well-equipped to deal with this issue as, by design, communication with external processes is rendered simple, and reinforced by the availability of a number of interfacing libraries for a number of different languages.

The Java API for kdb+ is a library which can be implemented easily within any Java application in order to interact with kdb+ processes. As with any application interface, its potential use cases are broad; in a scenario whereby a client wishes to introduce kdb+ gradually into a wider system, such an interface is essential for any interaction with Java processes, be they upstream or downstream. The straightforward implementation allows changes to legacy code to be lightweight, thus reducing the risk of wider system issues as kdb+ technology is introduced.

This paper illustrates how the Java API for kdb+ can be used to enable a Java program to interact with a kdb+ process. It will do so by first exploring the API itself in how it is structured and how it might be included in a development project. Examples will then be provided for core use cases for the API in a standard interfacing setup. Particular consideration will be given to how the API can facilitate subscription and publication to a kdb+ tickerplant process, a core component of any kdb+ tick capture system.

While https://code.kx.com/q/interfaces/java-client-for-q[1] should still be referred to as the primary source for up-to-date information on the Java API, the examples presented in this paper are designed to form a complementary set of practical templates. These templates can be combined and leveraged to facilitate the application of kdb+

1. https://code.kx.com/q/interfaces/java-client-for-q

across a broad range of problem domains, and are made available for reference and reuse at https://github.com/kxcontrib/javaapi[2].

2. https://github.com/kxcontrib/javaapi

API overview

The API is contained in a single source file, KxSystems/javakdb/src/kxc.java[3], which is available at the linked KxSystems Github repository. Inclusion in a given development project is, therefore, a straightforward matter of including the file with other source code under the package `kx`, and ensuring it is properly imported and referenced by other classes. If preferred, it can be compiled separately into a class or jar file to be included in the classpath for use as an external library or uploaded to a local repository for build integration.

As the API is provided as source, it is perfectly possible to further customize code to in order to meet specific requirements. However, this is not advised without either prior knowledge of how the interactions work or unless the solution to these requirements or issues are known. It is also possible, and in some contexts encouraged, to wrap the functionality of this class within a model suitable for your framework. An example of this might be the open-source qJava library which, while not compatible with the most recent kdb+ version at the time of writing, uses `c.java` as a core over which an object-oriented framework of q types and functionality has been applied.

The source file is structured as one outer class, `c`, within which are included a number of constants and inner classes which together model an environment for sending and receiving data from a kdb+ process. This section explores the fundamentals of the class in order to provide context and understanding of the practical use-cases for which the API can be employed.

Connection and interface logic

The highly-recommended means of connecting to a kdb+ process using the API is through the instantiation of the `c` object itself. There are three constructors provided for this purpose:

```
public c(String host,int port,String usernamepassword)
public c(String host,int port,String
usernamepassword,boolean useTLS)
public c(String host,int port)
```

These constructors are straightforward from a black-box perspective. The host and port are used to establish a socket object connection, with the username/password string serialized and passed to the remote instance for authorization. The core logic for all of these is the same; the host/port-only constructor attempts to retrieve the user

3. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java

string from the Java properties, and the constructor with the 'useTLS' boolean will, when flagged true, attempt to use an SSL socket instead of an ordinary socket.

It is also possible to set up the object to accept incoming connections from kdb+ processes rather than just making them. There are two constructors which, when passed a server socket reference, will allow a q session to establish a handle against the c object:

```
public c(ServerSocket s)
public c(ServerSocket s,IAuthenticate a)
```

`IAuthenticate` is an interface within the `c` class that can be implemented to emulate kdb+ server-side authentication, allowing the establishment of authentication rules similar to that which might be done through the kdb+ function .z.pw[4].

Both of these constructor families represent two 'modes' into which the `c` object can be instantiated. The first, and ultimately most widely used, is for making connections to kdb+ processes, which naturally would be used for queries, subscriptions and any task that requires the reception of or sending of data to said processes. The second, which sees Java act as the server, would see utility in management and aggregation of kdb+ clients, perhaps as a data sink or an intermediary interface for another technology.

Interactions between Java and kdb+ through these connections are largely handled by what might be called the 'k' family of methods in the `c` class. There are thirteen combined methods and overloads that fall under this group. They can be divided roughly into four groups:

Synchronous query methods

```
public Object k(String expr)
public Object k(String s,Object x)
public Object k(String s,Object x,Object y)
public void k(String s,Object x,Object y,Object z)
public synchronized Object k(Object x)
```

These methods are responsible for handling synchronous queries to a kdb+ process. The String parameter will represent either the entire q expression or the function name; in the case of the latter, the Object parameters may be used to pass values into that function. In all instances, the String/Object combinations are merged into a single object to be passed to the synchronized `k(Object)` method.

4. http://code.kx.com/q/ref/dotz/#zpw-validate-user

Asynchronous query methods

```
public void ks(String expr)
public void ks(String s,Object x)
public void ks(String s,Object x,Object y)
public void ks(String s,Object x,Object y,Object z)
public void ks(Object obj)
```

These methods are responsible for handling asynchronous queries to a kdb+ process. They operate logically in a similar manner to the synchronous query method, with the exception that they are, of course, void methods in that they neither wait for nor return any response from the process.

Incoming message method

```
public Object k()
```

This method waits on the class input stream and will deserialize the next incoming kdb+ message. It is used by the `c` synchronous methods in order to capture and return response objects, and is also used in server-oriented applications in order to capture incoming messages from client processes.

Response message methods

```
public void kr(Object obj)
public void ke(String text)
```

These methods are typically used in server-oriented applications to serialize and write response messages to the class output stream. `kr(Object)` will act much like any synchronous response, while `ke(String)` will format and output an error message.

The use of these constructors and methods will be treated in more practical detail through the use-case examples below.

Models and type mapping

The majority of q data types are represented in the API through mapping to standard Java objects. This is best seen in the method `c.r()`[5], which reads bytes from an incoming message and converts those bytes into representative Java types.

The full list of Java type mappings[6] can be found on the code.kx website.

Basic types

The method `c.r()` uses deciphered bytes within a certain range to point to further methods which return the appropriate typed object. These are largely self-explanatory, such as booleans and integer primitives mapping directly to one another, or q UUIDs mapping to `java.util.UUID`. There are some types with caveats, however:

- The kdb float type (9) maps to `java.lang.Double` and *not* `java.lang.Float`, which matches to the kdb real type (8).
- Java strings map to the kdb symbol type (11). In terms of reading or passing in data, this means that passing `"String"` from Java to kdb would result in `` `String``. Conversely, passing `"String"` (type 10 list) from kdb to Java would result in a six-index character array.

Time-based types

Of particular interest is how the mapping handles temporal types, of which there are eight:

q type	*id*	*Java type*	*note*
datetime	15	`java.util.Date`	This Java class stores times as milliseconds passed since the Unix epoch. Therefore, like the q datetime, it can represent time information accurate to the millisecond. (This is in spite of the default output format of the class).
date	14	java.sql.Date	While this Java class extends the `java.util` date object it is used specifically for the date type as it restricts usage and output of time data.
time	19	`java.sql.Time`	This also extends `java.util.Date`, restricting usage and output of date data this time.

5. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L709

6. http://code.kx.com/q/interfaces/java-client-for-q/#type-mapping

q type	*id*	*Java type*	*note*
timestamp	12	`java.sql.Timestamp`	This comes yet again from the base date class, extended this time to include nanoseconds storage (which is done separately from the underlying date object, which only has millisecond accuracy). This makes it directly compatible with the q timestamp type.
month	13	inner class `c.Month`[7]	
timespan	16	inner class `c.Timespan`[8]	
minute	17	inner class `c.Minute`[9]	
second	18	inner class `c.Second`[10]	

When manipulating date, time and datetime data from kdb+ it is important to note that while `java.sql.Date` and `Time` extend `java.util.Date`, and can be assigned to a `java.util` reference, that many of the methods from the original date class are overridden in these to throw exceptions if invoked. For example, in order to create a single date object for two separate SQL `Date` and `Time` objects, a `java.util.Date` object should be instantiated by adding the `getTime()` values from both SQL objects:

```
//Date value = datetime - time
java.sql.Date sqlDate = (java.sql.Date)qconn.k(".z.d");
// Time value - datetime - date
java.sql.Time sqlTime = (java.sql.Time)qconn.k(".z.t");
java.util.Date utilDate= new java.util.Date(sqlDate.getTime()+sqlTime.getTime());
```

The four time types represented by inner classes are somewhat less prevalent than those modeled by `Date` and its subclasses. These classes exist as comparable models due to a lack of a clear representative counterpart in the standard Java library, although their modeling is for the large part fairly simple and the values can be easily implemented or extracted.

Dictionaries and tables

Kdb+ dictionaries (type 99) and tables (type 98) are represented by the internal classes `Dict` and `Flip` respectively. The makeup of these models is simple but effective, and useful in determining how best to manipulate them.

7.
https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L300
8.
https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L376
9.
https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L326
10.
https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L351

The `Dict` class[11] consists of two public `java.lang.Object` fields (`x` for keys, `y` for values) and a basic constructor, which allows any of the represented data types to be used. However, while from a Java perspective any object could be passed to the constructor, dictionaries in q are always structured as two lists. This means that if the object is being created to pass to a q session directly, the `Object` fields in a `Dict` object should be assigned arrays of a given representative type, as passing in an atomic object will result in an error.

For example, the first of the following dictionary instantiation is legal with regards to the Java object, but because the pairs being passed in are atomic, it would signal a type error in q. Instead, the second example should be used, and can be seen as mirroring the practice of enlisting single values in q:

```
new c.Dict("Key","Value"); // not q-compatible
new c.Dict(new String[] {"Key"}, new String[] {"Value"}); // q-compatible
```

As the logical extension of that, in order to represent a list as a single key or pair, multi-dimensional arrays should be used:

```
new c.Dict(new String[] {"Key"}, new String[][] {{"Value1","Value2","Value3"}});
```

Flip (table) objects[12] consist of a String array for columns, an object array for values, a constructor and a method for returning the object array for a given column. The constructor takes a dictionary as its parameter, which is useful for the conversion of one to the other should the dictionary in question consist of single symbol keys. Of course, with the fields of the class being public, the columns and values can be assigned manually.

Keyed tables in q are dictionaries in terms of type, and therefore will be represented as a `Dict` object in Java. The method `td(Object)`[13] will create a `Flip` object from a keyed table `Dict`, but will remove its keyed nature in the process.

GUID

The globally unique identifier (GUID) type was introduced into kdb+ with version 3.0 for the purpose of storing arbitrary 16-byte values, such as transaction IDs. Storing such values in this form allows for savings in tasks such as memory and storage usage, as well as improved performance in certain operations such as table lookups when compared with standard types such as Strings.

11. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L427

12. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L440

13. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L1396

Java has its own unique identifier type: `java.util.UUID` (universally unique identifier). In the API the kdb+ GUID type maps directly to this object through the extraction and provision of its most and least significant long values. Otherwise, the only high-level difference in how this type can be used when compared to other types handled by the API is that a `RuntimeException` will be thrown if an attempt is made to serialize and pass a UUID object to a kdb+ instance with a version lower than 3.0.

More information on these identifier types can be found in the Kx documentation[14] as well as the core Java documentation[15].

Null types

Definitions for q null type representations in Java are held in the static `Object` array `NULL`, with index positions representing the q type.

```
public static Object[] NULL={
    null,
    new Boolean(false),
    new UUID(0,0),
    null,
    new Byte((byte)0),
    new Short(Short.MIN_VALUE),
    new Integer(ni),
    new Long(nj),
    new Float(nf),
    new Double(nf),
    new Character(' '),
    "",
    new Timestamp(nj),
    new Month(ni)
    ,new Date(nj),
    new java.util.Date(nj),
    new Timespan(nj),
    new Minute(ni),
    new Second(ni),
    new Time(nj)
};
```

Of note are the integer types, as the null values for these are represented by the minimum possible value of each of the Java primitives. Shorts, for example, have a minimum value of -372768 in Java, but a minimum value of -372767 in q. The extra negative value in Java can therefore be used to signal a null value to the q connection logic in the `c` class.

Float and real nulls are both represented in Java by the `java.lang.Double.NaN` constant. Time values, essentially being longs under the bonnet, are represented by

14. http://code.kx.com/q/ref/datatypes/#guid

15. https://docs.oracle.com/javase/7/docs/api/java/util/UUID.html

the same null value as longs in Java. Month, minute, second and timespan, each with custom model classes, use the same null value as ints.

The method `c.qn(Object)`[16] can assist with checking and identifying null value representations, as it will check both the `Object` type and value against the `NULL` list.

It is worth noting that infinity types are not explicitly mapped in Java, although kdb+ float and real infinities will correspond with the infinity constants in `java.lang.Double` and `java.lang.Float` respectively.

Exceptions

`KException`[17] is the single custom exception defined and thrown by the API. It is fairly safe to assume that a thrown `KException` denotes a q error signal, which will be included in the exception message when thrown.

Other common exceptions thrown in the API logic include:

IOException

> Denotes issues with connecting to the kdb+ process. It is also thrown by `c.java` itself for such issues as authentication.

RuntimeException

> Thrown when certain type implementations are attempted on kdb+ versions prior to their introduction (such as the GUIDs prior to kdb+ 3.0)

UnsupportedEncodingException

> It is possible, through the method `setEncoding`, to specify character encoding different to the default (`ISO-859-1`). This exception will be thrown commonly if the default is changed to a charset format not implemented on the target Java platform.

16. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L1355

17. https://github.com/KxSystems/javakdb/blob/master/src/kx/c.java#L457

Practical use-case examples

The following examples consist of common practical tasks that a Java developer might be expected to carry out when interfacing with kdb+. The inline examples take the form of extracted sections of key logic and output, and are available as example classes from the Kx repository 'FIXME' so that they may be used as starting points or templates. Tthe points at which the examples occur in the repository code are linked.

These examples assume, at minimum, a standard installation of 32-bit kdb+ on the local system, and a suitable Java development environment.

Connecting to a kdb+ process

Starting a local q server

During development, it can be helpful to start a basic q server to which a Java process can connect. This requires the opening of a port, for which there are two basic methods:

Example: Starting q with `-p` parameter

```
$ q -p 10000


q)\p // command to show the port that q is listening on
10000i
```

Example: Using the `\p` system command

```
$ q


q)\p 10000 // set the listening port to 10000
q)\p
10000i
```

To close the port, it should be set to its default value of 0 i.e. `\p 0`.

Setting up a q session in this manner will allow other processes to open handles to it on the specified port. The remainder of the examples in this paper assume an opened q session listening on port 10000, with no further configuration unless otherwise specified.

Opening a socket connection

As discussed in the previous section, the `c` class establishes connections via its constructors.

For connecting to a listening q process, one useful mechanism might be to create a factory class with a method that returns a connected `c` object based on what is passed to it. This way, any number of credential combinations can be set whilst allowing the creation of multiple connections, say for reconnection purposes:

Example: `QConnectionFactory.java`

```
public QConnectionFactory(String host, int port,
    String username, String password, boolean useTLS) {
  this.host=host;
  this.port=port;
  this.username=username;
  this.password=password;
  this.useTLS=useTLS;
}

//[…]

public c getQConnection() throws KException, IOException {
  return new c(host,port,username+":"+password,useTLS);
}
```

These constructors will always return a `c` object connected to the target session, and failure to do so will result in a thrown exception; `IOException` will denote the port not being open or available, and a `KException` will denote something wrong with the q process itself (such as `'access` for incorrect or incomplete credentials).

For the remaining examples, connections will be made using a custom `QConnectionFactory` object returned from a static method `getDefault()`, which will instantiate the object with the host `localhost` and the port 10000:

Example: `QConnectionFactory.java`

```
public static QConnectionFactory getDefault() {
  return new QConnectionFactory("localhost", 10000);
}
```

Connection objects created using this will be given the variable name `qConnection` unless otherwise stated.

Running queries using k methods

Queries can be made using the ‘k’ family of methods in the `c` class. For synchronous queries, that might be used to retrieve data (or, more generally, to halt execution of the Java process until a response is received), the k methods with parameter combinations of strings and objects might be used. For asynchronous queries, as might be used in a feed-handler process to push data to a tickerplant, the `ks` methods would be used.

The methods `k()`, `kr()` and `ke()` would not see explicit use in the querying of a server q process, but are more significant when the Java process acts as the server, as will be touched upon below.

The following examples demonstrate some of the means by which these synchronous and asynchronous queries may be called:

Example: `SimpleQueryExamples.java`

```
//Object for storing the results of these queries
Object result = null;

//Basic synchronous q expression
result = qConnection.k("{x+y}\[4;3\]");
System.out.println(result.toString());

//parameterised synchronous query
result = qConnection.k("{x+y}",4,3); //Note autoboxing!
System.out.println(result.toString());

//asynchronous assignment of function
qConnection.ks("jFunc:{x-y+z}");

//synchronous calling of that function
result = qConnection.k("jFunc",10,4,3);
System.out.println(result);

//asynchronous error - note no exception can be returned, so be careful!
qConnection.ks("{x+y}\[4;3;2\]");

//Always close resources\!
qConnection.close();
```

Extracting data from returned Objects

Note on internal variables and casting

The relationship between the kdb+ types and their Java counterparts has been discussed in the previous section. From a practical perspective, it is important to note that almost all objects and fields that might return from a given synchronous query will be of type `Object`, and will therefore more often than not require casting in order to be manipulated properly. Care must be taken, therefore, to ensure that the types that can be returned from a given query are known and handled appropriately so as to avoid unwanted exceptions.

The exception to this might be the column names of a `flip` object (once cast itself) held in the field `flip.x`. This field is already typed as `String[]`, as column names must always be symbols in q.

Kdb+ types that map to primitives (such as int) can be passed in Java to a `k` method as a primitive thanks to autoboxing[19], but will always be returned as the corresponding wrapper object (such as `Integer`).

Extracting atoms from a list

Lists will always be returned as an array of the given list type, or as `Object[]` if the list is generic. Extraction of atomic values from a list, therefore, is as simple as casting the return object to the appropriate array type and accessing the desired index:

Example: `ExtractionExamples.java`

```
//Get a list from the q session
Object result = qConnection.k("(1 2 3 4)");

//Cast the returned Object into long[], and retrieve the desired result.
long[] castList = ((long[]) result);
long extractedAtom = castList[0];
System.out.println(extractedAtom);
```

If the type of list is unknown, the method `c.t(Object)` can be used to derive the q type of the object, and theoretically could be useful in further casting efforts.

Extracting lists from a nested list

Accessing a list from a nested list is similar to accessing a value from any list. Here there are two casts required: a cast to `Object[]` for the parent list and then again to the appropriate typed array for the extracted list:

Example: `ExtractionExamples.java`

```
// Start by casting the returned Object into Object[]
Object[] resultArray = (Object[]) qConnection.k("((1 2 3 4); (1 2))");

//Iterate through the Object array
for (Object resultElement : resultArray) {

  //Retrieve each list and cast to appropriate type
  long[] elementArray = (long[]) resultElement;

  //Iterate through these arrays to access values.
  for(long elementAtom : elementArray) {
    System.out.println(elementAtom);
  }
}
```

19. https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

Working with dictionaries

The `Dict` inner class is used for all returned objects of q type dictionary (and therefore, by extension, keyed tables). Key values are stored in the field `Dict.x`, and values in `Dict.y`, both of which will generally be castable as an array.

Aside from matching the index positions of `x` and `y`, there is no intrinsic key-value pairing between the two, meaning that alteration of either of the array structures can compromise the key-value relationship. The following example illustrates operations that might be performed on a returned dictionary object:

Example: `ExtractionExamples.java`

```
//Retrieve Dictionary
c.Dict dict = (c.Dict) qConnection.k("`a`b`c!((1 2 3);\"Second\"; (`x`y`z))");
//Retrieve keys from dictionary
String[] keys = (String[]) dict.x;
System.out.println(Arrays.toString(keys));
//Retrieve values
Object[] values = (Object[]) dict.y;
//These can then be worked with similarly to nested lists
long[] valuesLong = (long[]) values[0];
//[…]
```

Working with tables

The inner class `c.Flip` used to represent tables operates in a similar manner to `c.Dict`. The primary difference, as previously mentioned, is that `Flip.x` is already typed as `String[]`, while `Flip.y` will still require casting. The following example shows how the data from a returned `Flip` object might be used to print the table to console:

Example: `ExtractionExamples.java`

```
// (try to load trade.q first for this (create a table manually if not possible)
qConnection.ks("system \"l trade.q\"");
//Retrieve table
c.Flip flip = (c.Flip) qConnection.k("select from trade where sym = `a");

//Retrieve columns and data
String[] columnNames = flip.x;
Object[] columnData = flip.y;
//Extract row data into typed arrays
java.sql.Timestamp[] time = (java.sql.Timestamp[]) columnData[0];
String[] sym = (String[]) columnData[1];
double[] price = (double[]) columnData[2];
int[] size = (int[]) columnData[3];
int rows = time.length;

//Print the table now - columns first:
for (String columnName : columnNames)
{
  System.out.print(columnName + "\t\t\t");
}
System.out.println("\n----------------------------------------------------");
//Then rows:
for (int i = 0; i < rows; i++)
{
  System.out.print(time[i]+"\t"+sym[i]+"\t\t\t"+price[i]+"\t\t\t"+size[i]+"\n");
}
```

Creating and passing data objects

When passing objects to q via the `c` class, there is less emphasis on how a given object is created. Rather, such an operation is subject to the common pitfalls associated with passing values to a q expression; those of type and valence.

The k family of methods, regardless of its return protocol, will take either the `String` of a q expression or the `String` of a q operator or function, complemented by `Object` parameters. Given the nature of q as an interpreted language, all of these are serialized and sent to the q session with little regard for logical correctness.

It is important, therefore, that any expressions passed to a query method are syntactically accurate and refer to variables that actually exist in the target session. It is also important that any passed objects are mapped to a relevant q type, and function within the context that they are sent. `KException` messages to look out for while implementing these operations are `'type` and `'rank`, as these will generally denote basic type and valence issues respectively.

Creating and passing a simple list

The following method might be applied to all direct type mappings in the API; for simple lists (lists in which all elements are of the same type), it is enough to pass a Java array of the appropriate type.

The following example invokes the q `set` function, which allows for the passing of a variable name as well as an object with which the variable might be set:

Example: `CreateAndSendExamples.java`

```
//Create typed array
int[] simpleList = {10, 20, 30};
//Pass array to q using set function.
qConnection.k("set", "simpleList", simpleList)
```

Creating and passing a mixed list

Mixed lists should always be passed to kdb+ through an Object array, `Object[]`. This array may then hold any number of mapped types, including, if appropriate, other typed or Object arrays:

Example: `CreateAndSendExamples.java`

```
//Create generic Object array.
Object[] mixedList = {new String[] {"first", "second"}, new double[] {1.0, 2.0}};
//Pass to q in the same way as a simple list.
qConnection.k("set", "mixedList", mixedList);
```

Creating and passing dictionaries

`c.Dict` objects are instantiated by setting its `x` and `y` objects in the constructor, and these objects should always be arrays. Once created, the Dict can be passed to kdb+ like any other object:

Example: `CreateAndSendExamples.java`

```
//Create keys and values
Object[] keys = {"a", "b", "c"};
int[] values = {100, 200, 300};
//Set in dict constructor
c.Dict dict = new c.Dict(keys, values);
//Set in q session
qConnection.k("set","dict",dict);
```

Creating and passing tables

`c.Flip` objects are created slightly differently; it is best to instantiate these by passing a `c.Dict` object into the constructor. This is because tables are essentially collections of dictionaries in kdb+, and therefore using this constructor helps ensure that the Flip object is set up correctly.

It is worth noting that for this method to work correctly, the passed Dict object must use String keys, as these will map into the Flip object's typed `String[]` columns:

Example: `CreateAndSendExamples.java`

```
//Create rows and columns
int[] values = {1, 2, 3};
Object[] data = new Object[] {values};
String[] columnNames = new String[] {"column"};
//Wrap values in dictionary
c.Dict dict = new c.Dict(columnNames, data);
//Create table using dict
c.Flip table = new c.Flip(dict);
//Send to q using 'insert' method
qConnection.ks("insert", "t1", table);
```

Creating and passing GUID objects

Globally universal identifier objects are represented in Java by `java.util.UUID` objects, and are passed to kdb+ in an identical manner as other basic types. The JjJava object has a useful static method for generating random identifiers, which further streamlines this process and can see utility in some use cases where only a certain number of arbitrary identifiers are required:

Example: `CreateAndSendExamples.java`

```
//Generate random UUID object
java.util.UUID uuid = java.util.UUID.randomUUID();
System.out.println(uuid.toString());

//Pass object to q using set function
qConnection.k("set","randomGUID",uuidj);
System.out.println(qConnection.k("randomGUID").toString());
```

Of course, it should be remembered that kdb+ version 3.0 or higher is required to work with GUIDs, and running the above code connected to an older version will cause a `RuntimeException` to be thrown.

Reconnecting to a q process automatically

Requirements will often dictate that while q processes will need to be bounced (such as for End-of-Day processing), that a Java process will need to be able to handle loss and reacquisition of said processes without being restarted itself. A simple example might be a graphical user interface, where the forced shutdown of the entire application due to a dropped connection, or the lack of ability to reconnect, would be very poor design indeed.

Use of patterns such as factories can help with the task of setting up a reconnection mechanism, as it allows for the simple creation of a preconfigured object. For `c` Objects, given that they connect on instantiation, means that a connection can be re-established simply by calling the relevant factory method.

In order to handle longer periods of potential downtime, either loops or recursion should be used. The danger with recursive methodology here is that, given an extended without a timeout limitation, there is a risk of overflowing the method-call stack, as each failed attempt will invoke a new method onto the stack.

For mechanisms that may need to wait indefinitely, it might be considered safer to use an indefinite while-loop that makes use of catch blocks, continue and break statements. This averts the danger of `StackOverflowError` occurring and is easily modified to implement a maximum number of tries:

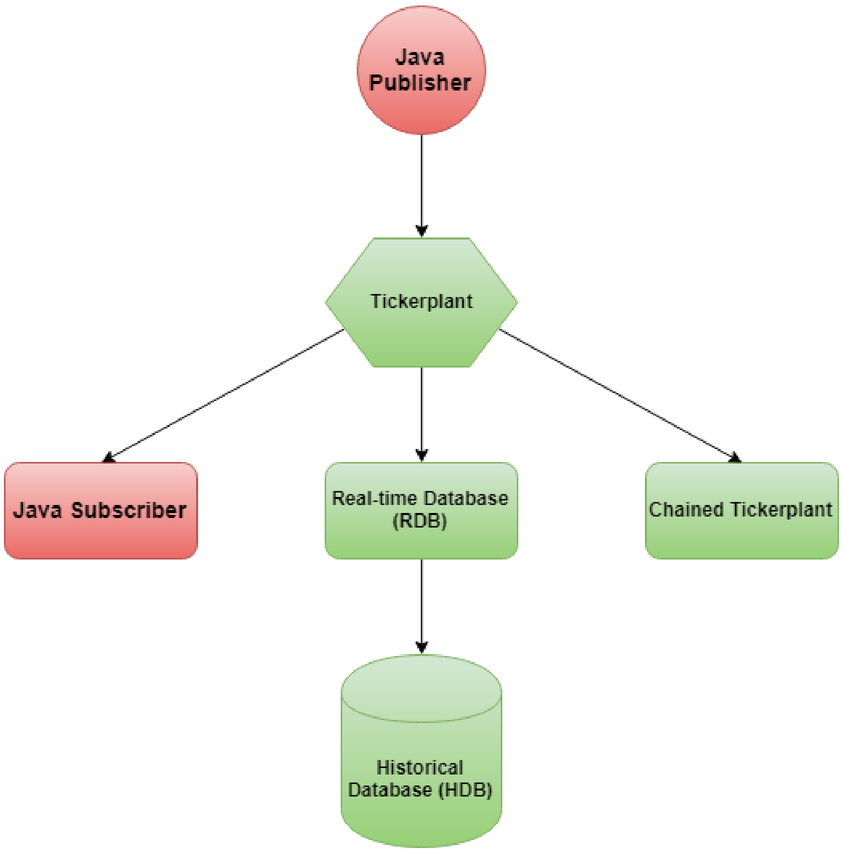
Example: `ReconnectionExample.java`

```
//initiate reconnect loop (possibly within a catch block).
while (true) {
  try {
    System.err.println("Connection failed - retrying..");
    //Wait a bit before trying to reconnect
```

```
      Thread.sleep(5000);
      qConnection = qConnFactory.getQConnection();
      System.out.println("Connection re-established! Resuming..");
      //Exit loop
      break;
    } catch (IOException | KException e1) {
      //resume loop if it fails
      continue;
    }
    …
  }
```

Kdb+ tickerplant overview

A kdb+ tickerplant is a q process specifically designed to handle incoming high-frequency data feeds from publishing process. Its primary responsibility is the management of subscription requests and the fast publication of data to subscribers. The following diagram illustrates a simple dataflow of a potential kdb+ tick system:



Simple dataflow of a potential kdb+ tick system

For further information regarding the above vanilla setup, see the whitepaper: *Building Real-time Tick Subscribers*[20].

20. http://code.kx.com/q/wp/building_real_time_tick_subscribers.pdf

Of interest in this whitepaper are the Java publisher and subscriber processes. As the kdb+ tick system is very widely used, both of these kinds of processes are highly likely to come up in development tasks involving kdb+ interfacing.

Test tickerplant and feedhandler setup

To facilitate the testing of Java subscriber processes we can implement example q processes freely available in the Kx repository. Simulation of a tickerplant can be achieved with `tick.q`[21]; Trade data, using the trade schema defined in `sym.q`, can then be published to this tickerplant using the definition for the file `feed.q` given here:

```
// q feed.q / with a default port of 5010 and default timer of 1000
// q feed.q -port 10000 / with a default timer of 1000
// q feed.q -port 10000 -t 2000

tph:hopen $[0=count .z.x;5010;"J"$first .Q.opt\[.z.x]`port]
if[not system"t";system"t 1000"]

publishTradeToTickerPlant:{
  nRows:first 1?1+til 3;
  tph(".u.upd";`trade;(nRows#.z.N;nRows?`IBM`FB`GS`JPM;nRows?150.35;nRows?1000));
  }

.z.ts:{
  publishTradeToTickerPlant[];
  }
```

The tickerplant and feed handlers can then be started by executing the following commands consecutively:

```
$ q tick.q sym -t 2000
$ q feed.q
```

Once the feedhandler is publishing to the tickerplant, processes can connect to it in order either to publish or subscribe to it.

Tickerplant subscription

==FIXME==

Extracting the table schema

Typical subscriber processes are required to make an initial subscription request to the tickerplant in order to receive data. See the publish and subscribe[22] cookbook article for details. This request involves calling the `.u.sub` function with two

21. https://github.com/KxSystems/kdb-tick/blob/master/tick.q

22. https://code.kx.com/q/cookbook/publish-subscribe

parameters. The first parameter is the table name and the second is a list of symbols for subscription. (Specifying a backtick in any of the parameters means all tables and/or all symbols).

Example: `TickSubscriberExample.java`

```
// Run sub function and store result
Object[] response = (Object[]) qConnection.k(".u.sub[`trade;`]");
```

If the `.u.sub` function is called synchronously, the tickerplant will return the table schema. If subscribing to one table, the returned object will be a generic Object array, with the table name in `object[0]` and a `c.Flip` representation of the schema in `object[1]`:

Example: `TickSubscriberExample.java`

```
// first index is table name
System.out.println("table name: " + response[0]);

// second index is flip object
c.Flip table = (c.Flip) response[1];

// Retrieve column names
String[] columnNames = table.x;
for (int i = 0; i < columnNames.length; i++) {
  System.out.printf("Column %d is named %s\n", i, columnNames[i]);
}
```

If more than one table is being subscribed to, the returned object will be an Object array consisting of the above object arrays; therefore, in order to retrieve each individual Flip object, this should be iterated against:

Example: `TickSubscriberExample.java`

```
// Run sub function and store result
Object[] response = (Object[]) qConnection.k(".u.sub[`;`]");

// iterate through Object array
for (Object tableObjectElement : response) {

  // From here, it is similar to the one-table schema extraction
  Object[] tableData = (Object[]) tableObjectElement;
  System.out.println("table name: " + tableData[0]);
  c.Flip table = (c.Flip) tableData[1];
  String[] columnNames = table.x;
  for (int i = 0; i < columnNames.length; i++) {
    System.out.printf("Column %d is named %s\n", i, columnNames[i]);
  }
}
```

Subscribing to a tickerplant data feed

Upon calling `.u.sub` and retrieving the schema, the tickerplant process will start to publish data to the Java process. The data it sends can be retrieved through the parameter-free `k()` method, which will wait for a response and return an Object (a `c.Flip` of the passed data) on publication:

Example: `TickSubscriberExample.java`

```
while (true) {

  //wait on k()
  Object response = qConnection.k();

  if(response != null) {
    Object[] data = (Object[]) response;

    //Slightly different.. table is in data[2]\!
    c.Flip table = (c.Flip) data[2];
    //[…]
  }
}
```

With the data in this form it can be manipulated in a number of meaningful ways. To iterate through the columns, `c.n` can be called on individual `flip.y` columns in order to provide a row count:

Example: `TickSubscriberExample.java`

```
String[] columnNames = table.x;
Object[] columnData = table.y;

//Get row count for looping
int rowCount = c.n(columnData[0]);

//Print out the table!
System.out.printf("%s\t\t\t%s\t%s\t%s\n",
    columnNames[0], columnNames[1], columnNames[2], columnNames[3]);
System.out.println("--------------------------------------------");
for (int i = 0; i < rowCount; i++) {

  //[Printing logic]

}
```

This mechanism might be then enveloped in an indefinite loop, such as a `while(true)` loop. Each iteration waits on the `k()` method returning published data, which will continue until one of the contributing processes fails (at which point an exception is caught and handled appropriately).

Tickerplant publishing

Publishing data to a tickerplant is almost always a necessity for a kdb+ feed-handler process. Java, as a common language of choice for third-party API development (e.g. Reuters, Bloomberg, MarkIT), is a popular language for feedhandler development, within which `c.java` is used to handle the asynchronous invocation of a publishing function.

Publishing rows

In general, publishing values to a tickerplant will require an asynchronous query much like the following:

```
qConnection.ks(".u.upd", "trade", data); //Where data is an Object[]
```

The parameters for this can be defined as follows:

The update function name (`.u.upd`)

> This is the function executed on the tickerplant which enables the data insertion. As per the norm with this API, this is passed as a string.

Table name

> A String representation of the name of the table that receives the data.

Data

> An object that will form the row(s) to be appended to the table. This parameter is typically passed as an object array, each index representing a table column.

In order to publish a single row to a tickerplant, typed arrays consisting of single values might be instantiated. These are then encapsulated in an object array and passed to the `ks` method:

Example: `TickPublisherExamples.java`

```
//Create typed arrays for holding data
String[] sym = new String[] {"IBM"};
double[] bid = new double[] {100.25};
double[] ask = new double[] {100.26};
int[] bSize = new int[]{1000};
int[] aSize = new int[]{1000};
//Create Object[] for holding typed arrays
Object[] data = new Object[] {sym, bid, ask, bSize, aSize};
//Call .u.upd asynchronously
qConnection.ks(".u.upd", "quote", data);
```

Publishing multiple rows is then just a case of increased length of each of the typed arrays:

Example: `TickPublisherExamples.java`

```
String[] sym = new String[] {"IBM", "GE"};
double[] bid = new double[] {100.25, 120.25};
double[] ask = new double[] {100.26, 120.26};
int[] bSize = new int[]{1000, 2000};
int[] aSize = new int[]{1000, 2000};
```

In order to maximize tickerplant throughput and efficiency, it is generally recommended to publish multiple rows in one go. For more information on this see the whitepaper *Kdb+tick Profiling for Throughput Optimization*[23].

Care has to be taken here to ensure that all typed arrays maintain the same length, as failure to do so will likely result in a kdb-side type error. Such errors are especially troublesome when using asynchronous methods, which will not return `KExceptions` in the same manner as sync methods! It is also worth noting that the order of the typed arrays within the object array should match that of the table schema.

Adding a timespan column

It is standard tickerplant functionality to append a timespan column to each row received from a feed handler if not included with the data passed, which is used to record when the data was received by the tickerplant. It's possible for the publisher to create the timespan column to prevent the tickerplant from adding one:

Example: `TickPublisherExamples.java`

```
//Timespan can be added here
c.Timespan[] time = new c.Timespan[] {new c.Timespan()};
String[] sym = new String[] {"GS"};
double[] bid = new double[] {100.25};
double[] ask = new double[] {100.26};
int[] bSize = new int[]{1000};
int[] aSize = new int[]{1000};
//Timespan array is then added at beginning of Object array
Object[] data = new Object[] {time, sym, bid, ask, bSize, aSize};
qConnection.ks(".u.upd", "quote", data);
```

This might be done, for example, to allow the feedhandler to define the time differently than simply logging the time at which the tickerplant receives the data.

Connecting from kdb+ to a Java process

The examples thus far have emphasized interfacing between Java and kdb+ very much from the perspective of a Java client connecting to a kdb+ server, using the constructors relevant to this purpose. It is very much possible to reverse these roles using the `c(Serversocket)` constructor, which enables a Java process to listen for incoming kdb+ messages on the specified port.

23. http://code.kx.com/q/wp/kdbtick_profiling_for_throughput_optimization.pdf

While the use cases for this 'server' mode of operation are not as common as they might be for 'client' mode connections, it can most certainly be useful. One potential use case, explored here, is the enabling of communications between two separate kdb+ systems by means of an interim proxy connection.

Setting up a single-connection proxy

To set this up, two `c` objects are required; one instantiated using the 'server' mode constructor and the other with the 'client' mode constructor. The former will be used to listen to the incoming connection of one kdb+ process, with the latter being the connection through which any received messages are passed on:

Example: `SingleConnectionProxy.java`

```
//declare c objects
c incomingConnection, outgoingConnection;

// Establish outgoing connection first
System.out.println("Connecting to q server on port 10000..");
outgoingConnection = QConnectionFactory.getDefault().getQConnection();

//Then wait for incoming connection
System.out.println("Waiting for incoming connection on port 500..");
incomingConnection = new c(new ServerSocket(500));
```

In a manner similar to tickerplant subscription, the method `k()` (without parameters) can be used to wait on and listen to any connecting q session. In this example, the object is retrieved in this fashion and can be passed on wholesale through the client mode connection:

Example: `SingleConnectionProxy.java`

```
while(true) {

  //k() method will wait until the kdb+ process sends an object.
  Object incoming = incomingConnection.k();
  try {

    //return the response received from sending the incoming data
    // to the outgoing connection
    incomingConnection.kr(outgoingConnection.k(incoming));
  } catch(IOException | KException e) {
    //return error responses too
    incomingConnection.ke(e.getMessage());
  }
}
```

It is easy to imagine further uses for this interim process; for example, upon receiving a table, such a process might be able to append or derive further information to append to it (say from another third-party interface) before passing it on.

Setting up a multiple-connection proxy

In the above example proxy, the server `c` object is instantiated with a new ServerSocket being created in its constructor. This is acceptable in this instance because we cared only about the handling of one connection.

In general, ServerSocket objects should not be used in this manner, as they are designed to handle more than a single incoming connection. Instead, the ServerSocket should be passed as a reference. With the addition of some simple threading, an application capable of redirecting all incoming connections on a given port to a client q session:

Example: `MultipleConnectionProxy.java`

```
//Create server socket reference beforehand..
ServerSocket serverSocket = new ServerSocket(500);

// Establish outgoing connection
System.out.println("Connecting to q server on port 10000..");
c outgoingConnection = QConnectionFactory.getDefault().getQConnection();
System.out.println("Accepting incoming connections..");

//Set up connection loop
while(true) {
  //Create c object with reference to server socket
  final c incomingConnection = new c(serverSocket);

  //Create thread for handling this connection
  new Thread(new Runnable() {
    @Override
    public void run() {
      while(true) {
      //Logic in this loop is similar to single connection
      //[…]
      }
    }
  //Run thread and restart loop.
  }).start();
}
```

This will allow any number of connections to be made via proxy to the client q session, and as only one client connection has to be made for this to operate (more can be established if desired), the limit to the number of connections can be set by the proxy itself. As in any case where threading is used, take care that such a method does not enable race conditions or concurrency issues; if necessary, steps can be taken to reduce the risk of such operations, such as synchronized blocks and methods.

Conclusion

In summary, this document has covered a variety of topics concerning the mechanics and application of the `c.java` interface for kdb+. Of the workings and examples shown, the most common use case for this interface will be connecting to a q process, executing queries and functions and managing any result objects. However, this document has also displayed the versatile nature of `c.java` as a tool, providing a handful of solutions to a given problem and able to fulfill server as well as client functions.

The practical examples should also help demonstrate that tasks required as part of a standard kdb+ toolset can be handled easily from the perspective of both Java developers interfacing with kdb+ for the first time, or kdb+ developers who are required to venture into Java development, for example, to help complete development of a feed handler. The benefit of such interfaces is felt keenly through the common role of these developers in helping to reconcile longstanding applications with contemporary technologies, often to the benefit of both.