Titanium Mobile: API Reference

Titanium.Database Class
Titanium.Database.DB Class
Titanium.Database.ResultSet Class

appcelerator

API

October 13, 2010

Appcelerator, Inc. 444 Castro Street, Suite 818, Mountain View, California 94041

Contents

Overview

The top level database module is used for creating and accessing the in-application Database. This gives you low-level access to an SQLite database which can be used in very general ways to store and organize information in your application. Most of your interaction with a local database will be through raw SQL, which you run against your database with the Titanium.Database.DB.execute method.

SQLite implements an SQL database engine. This document covers Titanium's objects and methods you'll use when working with the database. But it doesn't cover the specifics of SQLite, or SQL in general. Information on these can be found in other documents:

• SQLite Documentation — http://www.sqlite.org/docs.html

• SQL Tutorial — http://www.w3schools.com/sql/default.asp

In general, there are several choices for storing data in an application, including both persistent and non-persistent storage.

Storage Type	Persistence
Variables	single application execution only
Object properties	single application execution only
Titanium.App.Properties	across application sessions
Database	across application sessions
Filesystem	stored on device

For non-persistent storage, you can use either properties of an object or application variables. Object properties can store key/value pairs, and are associated with an instantiation of an object. Most objects in Titanium have associated properties which you can alter according to the needs of your application. In addition, many objects can have custom properties attached to them, with your own key/value pairs. By contrast, variables aren't associated with a specific object; they follow scoping rules and can be global to your application, or limited to the scope of a method or function. The values of variables and object properties are lost when an application terminates.

With Titanium, an application has two kinds of persistent storage which persist between sessions of the application. You can use either an SQLite database, documented here, or the Titanium.App.Properties module for persistent application storage. It is appropriate to use Titanium.App.Properties for lightweight key/value persistent storage. Use Titantium.Database when you need a more general purpose SQL style database.

Finally, there is the filesystem for persistent storage on the device. Filesystem storage is not tied to a specific application, and can be accessed from any running application.

If you want to delete the persistent storage for an application, there are two ways to accomplish this. The typical way would be to delete the application from the mobile device (or emulator). When the application is reinstalled, any previous persistent storage for that application will be lost — from the point of view of the device, this looks like a different application. Another way to delete persistent storage would be from within the application at run time, by executing code to clear values from properties stored with Titanium.App.Properties, or to delete any existing data from a database.

This document covers the methods that implement the SQLite database for persistent storage in an application. A database has a name; there can be multiple databases with different names used in a single application.

Titanium.Database Class

This is a singleton class which will be automatically instantiated when you first use one of its methods.

Instance Methods Summary

Name	Description
install	Install a pre-populated database from the application Resources folder (at build time) and return a reference to the opened database. It is safe to call this method multiple times since this method will only install once if it doesn't already exist on the device.
open	Open a database. If it doesn't yet exist, create it.

Object Properties

This module has no properties.

Events

This module has no events.

Methods

install

Returns an instance of Titanium.Database.DB. Install a pre-populated database from the application Resources folder (at build time) and return a reference to the opened database. It is safe to call this method multiple times since this method will only install once if it doesn't already exist on the device.

Arguments

Name	Type	Description
path	string	The path (relative to the main application Resources folder at build time) to the db to install. This file must be in the SQLite 3 file format.
name	string	the name of the database

Return Type

object (Titanium.Database.DB)

Example

```
var coldDB = Titanium.Database.install('../coldDB.db', 'cold');
```

open

Returns an instance of Titanium.Database.DB. Open a database. If it doesn't yet exist, create it.

Arguments

Name	Type	Description
name	string	the name of the database

Return Type

object (Titanium.Database.DB)

Example

```
var coldDB = Titanium.Database.open('cold');
```

Titanium.Database.DB Class

An instance of this class will be returned by Titanium.Database.open or Titanium.Database.install.

Instance Methods Summary

Name	Description
close	Close the database and release resources from memory. Once closed, this instance is no longer valid and must no longer be used. You should generally close the database when you're not using it; it's ok to close a database and then reopen it later in the application.
execute	Execute an SQL statement against the database and return a ResultSet.
remove	Remove the database files for this instance. WARNING: this is a destructive operation and cannot be reversed. All data in the database will be lost upon calling this function. Use with caution.

Object Properties

Name	Type	Description
lastInsertRowId	int	the last row identifier by the last INSERT query
name	string	the name of the database
rowsAffected	int	the number of rows affected by the last query

Events

This object has no events.

Methods

close

Close the database and release resources from memory. Once closed, this instance is no longer valid and must no longer be used. You should generally close the database when you're not using it; it's ok to close a database and then reopen it later in the application.

Arguments

This function takes no arguments.

Return Type

void

Example

```
coldDB.close();
```

execute

Execute an SQL statement against the database and return a ResultSet.

Arguments

Name	Type	Description
sql	string	the SQL to execute
vararg	array,...	Zero or more optional variable arguments passed to this function or an array of objects to be replaced in the query using ? substitution.

Return Type

object

Example

```
coldDB.execute('INSERT INTO coldBeverages (beverage) VALUES(?)', 'Juice');
```

remove

Remove the database files for this instance. WARNING: This is a destructive operation and cannot be reversed. All data in the database will be lost upon calling this function. Use with caution.

Arguments

This function takes no arguments.

Return Type

void

Example

```
coldDB.remove();
```

Titanium.Database.ResultSet Class

An instance of this class is returned by invoking a database SQL execute. The ResultSet object allows you to iterate over the result(s) of a query, pulling data out of the query. After iterating over the results using the ResultSet object, it is important when you're finished to call the ResultSet's close method, as illustrated in the code samples.

Instance Methods Summary

Name	Description
close	Close the result set and release resources from memory. Once closed, this instance is no longer valid and must no longer be used.
field	retrieve a row value by field index
fieldByName	retrieve a row value by field name
fieldCount	return the number of columns in the result set
fieldName	return the field name for field index
isValidRow	return true if the row is a valid row
next	Iterate to the next row in the result set. Returns false if no more results are available

Object Properties

Name	Type	Description
rowCount	int	the number of rows in the result set
validRow	boolean	returns true if the current row is still valid

Events

This object has no events.

Methods

close

Close the result set and release resources from memory. Once closed, this instance is no longer valid and must no longer be used.

Arguments

This function takes no arguments.

Return Type

void

field

Retrieve a row value by field index.

Arguments

Name	Type	Description
index	int	column index (which is zero based)

Return Type

object

Example

```
var rowValue = myResultSet.field(1);
```

---

fieldByName

Retrieve a row value by field name.

Arguments

Name	Type	Description
name	string	column name from SQL query

Return Type

object

Example

```
var rowValue = myResultSet.fieldByName('beverage');
```

---

fieldCount

Return the number of columns in the result set.

Arguments

This function takes no arguments.

Return Type

int

Example

```
var numColumns = myResultSet.fieldCount();
```

---

fieldName

Return the field name for field index.

Arguments

Name	Type	Description
index	int	field name column index (which is zero based)

Return Type

string

Example

```
var myName = myResultSet.fieldName(2);
```

isValidRow

Return true if the row is a valid row.

Arguments

This function takes no arguments.

Return Type

boolean

Example

```
var myResultSet = coldDB.execute('SELECT * FROM coldBeverages');
while (myResultSet.isValidRow()) {
     // Do something that iterates over all rows
     .
     .
     myResultSet.next()
}
myResultSet.close();
```

next

Iterate to the next row in the result set. Returns false if no more results are available.

Arguments

This function takes no arguments.

Return Type

boolean

Example

```
var myResultSet = coldDB.execute('SELECT * FROM coldBeverages');
while (myResultSet.isValidRow()) {
     // Do something that iterates over all rows
```

```
    .
    .
    myResultSet.next()
}
myResultSet.close();
```

Sample Programs

Simple database example illustrating persistence with a TableView

This sample program creates a TableView, getting some of the row titles from a database. (The list of cold beverages comes from the database.) As you click the row "Delete one cold beverage", it changes the data in the database. However, the display doesn't change until you quit the app and then restart it, when you'll see the rows recreated using data from the underlying database. If you delete all the cold beverages, then the app will see that there are no entries in the database and it will recreate a default collection of cold beverages the next time it starts up.

This example works with iPhone and Android.

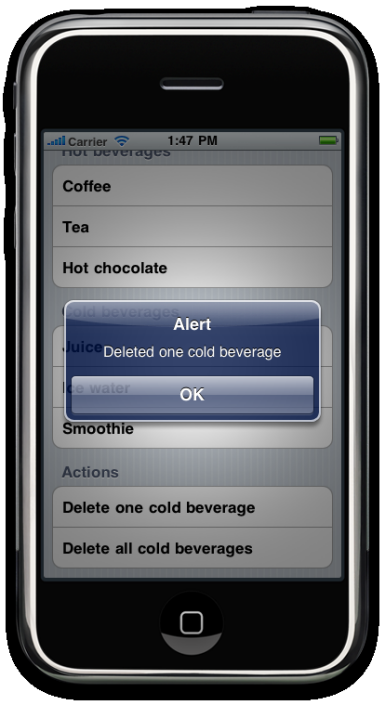
```
// app.js

// Create a window
var myWindow = Titanium.UI.createWindow({fullscreen:false});

// Create the first table view section, a collection of hot beverages
var maxRow = 0;
var hotTableViewSection = Titanium.UI.createTableViewSection({headerTitle:'Hot beverages'});
hotTableViewSection.add(Titanium.UI.createTableViewRow({title:'Coffee'}));
hotTableViewSection.add(Titanium.UI.createTableViewRow({title:'Tea'}));
hotTableViewSection.add(Titanium.UI.createTableViewRow({title:'Hot chocolate'}));
maxRow += 3;

// Create a second table view section, a collection of cold beverages, populated from a database
var coldTableViewSection = Titanium.UI.createTableViewSection({headerTitle:'Cold beverages'});

// First, create a database connection, creating the database if it doesn't yet exist
var coldDB = Titanium.Database.open('cold');

// Create the database table using an SQL command
// If it already exists, we will be using the existing data from a previous execution of the app
coldDB.execute(
     'CREATE TABLE IF NOT EXISTS coldBeverages (id INTEGER PRIMARY KEY, beverage TEXT)');

// Now, work with the database to create a second section of the TableView. This section will
// contain a list of cold beverages. First, we'll see if there are already any rows (beverages)
// in the database. If not, we'll add some. Then we'll use the database to populate the
// cold beverage section of the TableView.
// Try to read records from the database, and see if there are any records at all
var myResultSet = coldDB.execute('SELECT * FROM coldBeverages');
if (!myResultSet.isValidRow()) {
     // We have an empty database ...
     // Add 3 cold beverages to the database table
     coldDB.execute('INSERT INTO coldBeverages (beverage) VALUES(?)', 'Juice');
     coldDB.execute('INSERT INTO coldBeverages (beverage) VALUES(?)', 'Ice water');
     coldDB.execute('INSERT INTO coldBeverages (beverage) VALUES(?)', 'Smoothie');
}
```

```
myResultSet.close();

// Create a TableView row for each cold beverage in the database table
var myResultSet = coldDB.execute('SELECT * FROM coldBeverages');
while (myResultSet.isValidRow()) {
     coldTableViewSection.add(Titanium.UI.createTableViewRow({
         title:myResultSet.fieldByName('beverage')
     }));
     maxRow += 1;
     myResultSet.next();
}
myResultSet.close();// Finished with the result set
coldDB.close();// Finished with the database for now

// Create the third table view section, two delete buttons
var listTableViewSection = Titanium.UI.createTableViewSection({headerTitle:'Actions'});
listTableViewSection.add(Titanium.UI.createTableViewRow({title:'Delete one cold beverage'}));
listTableViewSection.add(Titanium.UI.createTableViewRow({title:'Delete all cold beverages'}));
maxRow += 2;

// Create a TableView; we'll add the row data later
var myTableView = Titanium.UI.createTableView({
     style:Titanium.UI.iPhone.TableViewStyle.GROUPED
});

// Add the hot and cold sections to the TableView, as well as the action buttons
myTableView.setData([hotTableViewSection, coldTableViewSection, listTableViewSection]);

// Add an event listener
myTableView.addEventListener('click', function(e) {
     // Reopen the database
     var coldDB = Titanium.Database.open('cold');

     if (e.index == (maxRow-2)) {
         // Delete one row from the database table
         // This will be seen after app quits and starts again
         var myResultSet = coldDB.execute('SELECT * FROM coldBeverages');
         if (myResultSet.isValidRow()) {
              coldDB.execute('DELETE FROM coldBeverages WHERE id = 1');
              Titanium.UI.createAlertDialog({
                   title:'Alert',
                   message:'Deleted one cold beverage'
              }).show();
         }
         myResultSet.close();
     }
     else if (e.index == (maxRow-1)) {
         // Delete all rows from the database table
         // This will be seen after app quits and starts again
         coldDB.execute('DELETE FROM coldBeverages');
         Titanium.UI.createAlertDialog({
              title:'Alert',
              message:'Deleted all cold beverages'
         }).show();
     }
     else {
         // Report which item was clicked
         Titanium.UI.createAlertDialog({
              title:'Alert',
```

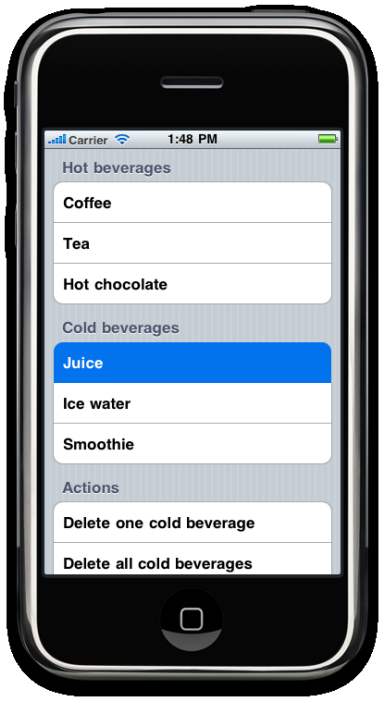
```
            message:'Click at index: '+e.index
        }).show();
        // Show that choice as selected
        if (Titanium.Platform.name != 'android') {
            myTableView.selectRow(e.index);
        }
    }
    coldDB.close();// Finished with the database for now
});

// Add the table to the window and open it
myWindow.add(myTableView);
myWindow.open();
```

Android Alerts

Android Buttons



iPhone Alerts



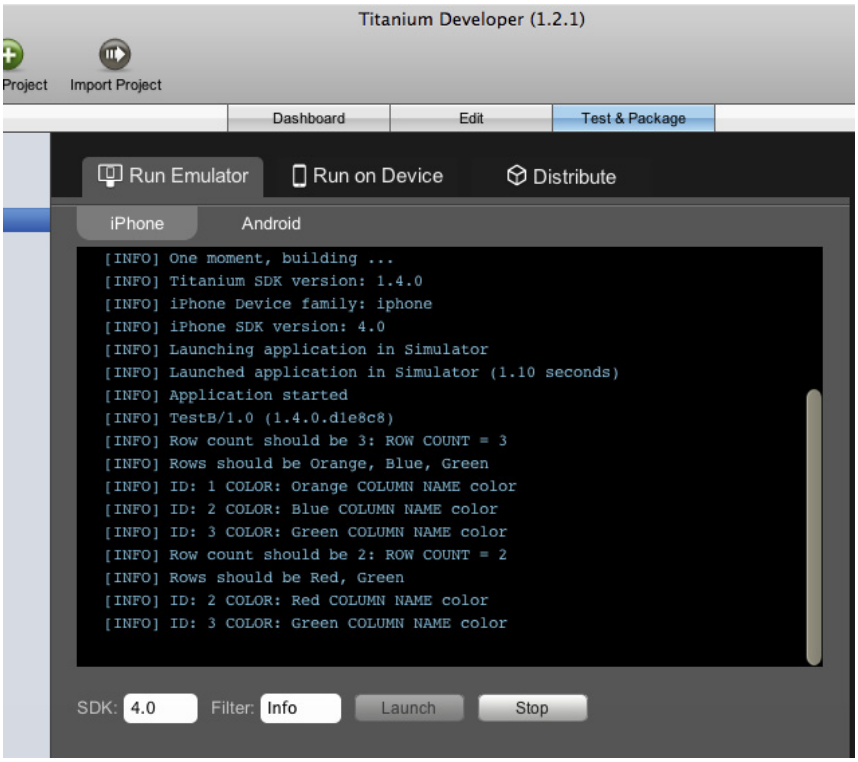
iPhone Buttons

Simple database example illustrating Create, Read, Update, Delete

This example shows the simple database create, read, update and delete (CRUD) operations of a database. This sample code replaces your app.js, but it doesn't implement a user interface. You can see messages indicating the results of the database operations in the Titanium Developer console window.

This example works with iPhone and Android.

```
// app.js

// Open (create if it doesn't exist) a database
var db = Titanium.Database.open ('simple');

// Create the table if necessary
db.execute('CREATE TABLE IF NOT EXISTS colorlist (ID INTEGER, COLOR TEXT)');

// Clear out any previous data from the table
db.execute('DELETE FROM colorlist');

// Add several values (rows) to the table
db.execute('INSERT INTO colorlist (ID, COLOR) VALUES(?,?)', 1, 'Orange');
db.execute('INSERT INTO colorlist (ID, COLOR) VALUES(?,?)', 2, 'Blue');
db.execute('INSERT INTO colorlist (ID, COLOR) VALUES(?,?)', 3, 'Green');

// Get all the rows, and iterate on them to report values of each
var rows = db.execute('SELECT * FROM colorlist');
Titanium.API.info('Row count should be 3: ROW COUNT = ' + rows.getRowCount());
Titanium.API.info('Rows should be Orange, Blue, Green');
while (rows.isValidRow()) {
     Titanium.API.info('ID: ' + rows.field(0) + ' COLOR: '
         + rows.fieldByName('color') + ' COLUMN NAME ' + rows.fieldName(1));
     rows.next();
}
rows.close();

// Now, update one of those rows
var updateName = 'Red';
var updateID = 2;
db.execute('UPDATE colorlist SET COLOR = ? WHERE ID = ?', updateName, updateID);

// Delete one of the rows
db.execute('DELETE FROM colorlist WHERE ID = ?', 1);

// Get all the rows, and iterate on them to report values of each
var rows = db.execute('SELECT * FROM colorlist');
Titanium.API.info('Row count should be 2: ROW COUNT = ' + rows.getRowCount());
Titanium.API.info('Rows should be Red, Green');
while (rows.isValidRow()) {
     Titanium.API.info('ID: ' + rows.field(0)+' COLOR: '
         + rows.fieldByName('color') + ' COLUMN NAME ' + rows.fieldName(1));
     rows.next();
}
rows.close();

// Close the database to conserve resources
db.close();
```



Console Output CRUD

Database example using blob data

This sample shows how the database can work with arbitrary data as a blob. However, it's generally not good practice to store large amounts of data, like images, in the database. A better strategy if you have lots of images, for example, would be to store the images as files, and then keep the corresponding filenames in the database.

This example works with iPhone and Android.

```
// app.js

// Create a window
var myWindow = Titanium.UI.createWindow({fullscreen:false});

// Get an image from a file
var imageFile = Titanium.Filesystem.getFile('myphoto.jpg');
var oneImage = imageFile.read();

// First, create a database connection, creating the database if it doesn't yet exist
var db = Titanium.Database.open('images');

// Create the database table using an SQL command
db.execute('CREATE TABLE IF NOT EXISTS pics (id INTEGER, image BLOB)');

// Starting with an empty table, insert the image into the table
// This is an example -- in general, you probably don't want to store images in the database
db.execute('DELETE FROM pics');
```

```
db.execute('INSERT INTO pics (id, image) VALUES(?,?)', 1, oneImage);

// Get every record from the database, and get the image that we just stored
// (Note that if you find lots of matching records, this could use up an
// enormous amount of memory, since all records would be loaded into memory.
// Therefore, consider what technique makes sense for your application.)
var myResultSet = db.execute('SELECT * FROM pics WHERE id=1');
if (myResultSet.isValidRow()) {
    retrievedImage = myResultSet.field(1);
}
myResultSet.close();

// Display the image that was retrieved from the database
var myImageView = Titanium.UI.createImageView({
     image:retrievedImage
});

// Add the ImageView to the Window, and open the Window
myWindow.add(myImageView);
myWindow.open();

// Close the database to conserve resources
db.close();
```

Blob image on iPhone

Database example using the JavaScript module design pattern

Here's an advanced example of using database operations in Titanium, taken from Kevin Whinnery's blog post. This example is in the form of the module JavaScript design pattern, which is presented here:

http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth

In this sample code, we define a JavaScript module which performs operations on our database. We'll implement the four standard database operations, Create, Read, Update and Delete (in database circles known by the acronym CRUD) in the form of the module pattern. This is an interesting application of the module pattern to create your own API for database operations that you can use in your own applications.

This sample code replaces your app.js, but it doesn't implement a user interface. You can see messages indicating the results of the database operations in the Titanium Developer console window. The beginning part of the program defines the new db function using the module pattern, and the tests that call db are at the end of the program.

When the module is initially created, it populates the database with the tables it needs, if they do not exist. Note that the backslash in the SQL string simply denotes a multi-line JavaScript string literal.

Here we have just a single table with TODO items in it. We also maintain a reference to our application database which we can use in all of our API functions. Our public API has operations to read, update, create and delete records in our database. When working with a ResultSet, you can get properties from the table row using fieldByName, as shown in the example.

This shows how you might implement database operations in a Titanium application. Whether you use the module pattern illustrated here, or another approach, you would typically want to separate the database logic into a JavaScript file that you could include in multiple windows. This way, all your database interaction is contained in one place.

Here's the sample code, replacing your app.js. This example works with iPhone and Android.

```
// app.js

var db = (function() {

  //create an object which will be our public API
  var api = {};

  //maintain a database connection we can use
  var db = Titanium.Database.open('todos.db');

  //Initialize the database
  db.execute('CREATE TABLE IF NOT EXISTS todos (id INTEGER PRIMARY KEY, item TEXT)');

  //This will delete all data from our table
  db.execute('DELETE FROM todos');

  //Create a to-do item - db.create(item)
  api.create = function(text) {
    db.execute('INSERT INTO todos (item) VALUES(?)',text);
    return db.lastInsertRowId; //return the primary key for the last insert
  };

  //List all to-do items - db.read()
  api.all = function() {
    var results = [];

    //Get to-do items from database
    var resultSet = db.execute('SELECT * FROM todos');
    while (resultSet.isValidRow()) {
      results.push({
        id: resultSet.fieldByName('id'),
        item: resultSet.fieldByName('item')
      });
        resultSet.next();
```

```
    }
    resultSet.close();

   //return an array of JavaScript objects reflecting the to-do items
    return results;
  };

  //Get a to-do item by a specific ID
  api.get = function(id) {
    var result = null;
    var resultSet = db.execute('SELECT * FROM todos WHERE id = ?', id);
    if (resultSet.isValidRow()) {
      result = {
        id: resultSet.fieldByName("id"),
        item: resultSet.fieldByName("item")
      };
    }
    resultSet.close();
    return result;
  };

  //Update an exisiting to-do item - db.update(item)
  api.update = function(todoItem) {
    db.execute("UPDATE todos SET item = ? WHERE id = ?",
     todoItem.item, todoItem.id);

    //return the number of rows affected by the last query
    return db.rowsAffected;
  };

  //Delete a to-do item - db.del(item)
  api.del = function(id) {
    db.execute("DELETE FROM todos WHERE id = ?", id);

   //return the number of rows affected by the last query
    return db.rowsAffected;
  };

  //return our public API
  return api;
}());


// ****************************
// BEGIN DATABASE TEST EXAMPLES
// ****************************

// CREATE AND GET
var lastInsert = db.create("Pick up the milk"); // this should create a to-do item
var ourTodo = db.get(lastInsert); //this should fetch the to-do item we just created
Titanium.API.info("Just created a to-do item (Should be 'Pick up the milk'): "+ourTodo.item);

// ALL
var anotherTodo = db.create("Conquer the world");
var all = db.all();
Titanium.API.info("Records in database (Should be 2): "+all.length);

// UPDATE
ourTodo.item = "Pick up Mountain Dew instead of milk";
```

```
var rowsAffected = db.update(ourTodo);
Titanium.API.info("Rows affected (Should be 1): "+rowsAffected);
ourTodo = db.get(ourTodo.id);
Titanium.API.info(
"Just got our updated record (Should be 'Pick up Mountain Dew instead of milk'): "+ourTodo.item);

//DELETE
db.del(ourTodo.id);
var stillThere = db.get(ourTodo.id);
Titanium.API.info(
"Just tried to get a record that's not there.  Result should be null: " +stillThere);
```

Conslole Output Blob Pattern

Revision History

9/02/2010	Initial release
9/14/2010	Added additional sample programs
10/13/2010	Revisions to sample code and minor text corrections