CMER

Centre for Mobile Education and Research

Data Management

Overview

- MIDI Record Stores
- Persistence Storage
- Manage Persistent Data
- Manage Custom Objects
- Using the MIDP Record Store
- Data Synchronization
- Support for Backup & Restore over Wireless Network
- SyncCollection

Storage Method: MIDI Record Stores

- The MIDP record store allows an application to be portable across multiple devices that are compatible with the Java Platform, Micro Edition.
- In MIDP, store data as records in *RecordStore* objects. MIDP records store data only as byte arrays.
- In MIDP 2.0 and later, if an application creates a record store using the *RecordStore.AUTHMODE_ANY* field, a MIDlet suite can share the record store with other MIDlet suites.

Persistence Model

- **A type of model that provides a flexible and efficient way to store data on BlackBerry devices**
- **Ability to save any *Object* in the persistent store. Results in faster searches for data in the persistent store rather than searching in the record store model.**
- **To store custom object types, the class of the custom type must use the Persistable interface.**
- **Using this model, applications can share data at the discretion of the application that creates the data.**

Persistence Storage

- Security
  - applications on the BlackBerry device that are digitally signed by Research In Motion can access the data in the persistent store.
- Administrative control
  - With the BES, system administrators can use IT policies to control the use of persistent storage by third-party applications.
- Data integrity
  - To maintain the integrity of, partial updates are not made if an error occurs during a commit. Data in the PersistentObject retains the values from the last commit in order to preserve data integrity.

Persistence Storage (Cont.)

- If the BlackBerry JVM performs an emergency garbage collection operation due to low memory, outstanding transactions are committed immediately to avoid compromising data integrity.
- If the device fails during this operation, partially completed transactions are committed when the BlackBerry device starts.
- Outstanding transactions are not committed during normal garbage collection operation.

Manage Persistent Data

- Create a unique long key
  - Each PersistentObject has a unique long key.
    - In the IDE, type a string value, such as *com.rim.samples.docs.userinfo.*
    - Select this string.
    - Right-click this string and click Convert ‘com.rim.samples.docs.userinfo’ to long.
    - Include a comment in your code to indicate the string that you used to generate the unique long key.
- Create a persistent data store
  - Create a single static *PersistentObject.*
  - Invoke *PersistentStore.getPersistentObject*, using the unique long key as a parameter.

```
static PersistentObject store;
    static {
        store = PersistentStore.getPersistentObject(
        0xa1a569278238dad2L );
    }
```

- Store an object persistently
  - Invoke *setContents()* on a *PersistentObject*. This method replaces existing content with the new content.

Manage Persistent Data (Cont.)

- To save the new content to the persistent store, invoke *commit().*

```
String[] userinfo = {username, password};
        synchronized(store) {
                store.setContents(userinfo);
                store.commit();
        }
```

- Store objects in a batch transaction
  - To use a batch transaction to commit objects to the persistent store,
    invoke *PersistentStore.getSynchObject().* This method retrieves the
    persistent store monitor that locks the object.
  - Synchronize on the object.
  - Invoke *commit()* as necessary. If any commit in the batch fails, the
    entire batch transaction fails.
- Commit a monitor object separately from a batch transaction.
  - Invoke forceCommit() while synchronizing the monitor object.

Manage Persistent Data (Cont.)

- Retrieve persistent data
  - Invoke getContents() on a PersistentObject.
  - To convert to your desired format, perform an explicit cast on the object that *PersistentObject.getContents()* returns.

```
synchronized(store) {
        String[] currentinfo = (String[])store.getContents();
        if(currentinfo == null) {
                Dialog.alert(_resources.getString(APP_ERROR));
        }
        else {
                currentusernamefield.setText(currentinfo[0]);
                currentpasswordfield.setText(currentinfo[1]);
        }
}
```

Manage Persistent Data (Cont.)

- Remove all persistent data from an Application.
  - If you delete the .cod file that defines a *PersistentStore*, then all persistent objects that the .cod file created are deleted.
  - Invoke *PersistentStore.destroyPersistentObject(),* providing as a parameter a unique key for the PersistentObject.
- Remove specific persistent data from a java application.
  - To delete individual data, treat the data as normal objects, and remove references to it. A garbage collected operation removes the data.

Manage Custom Objects

- Create an object to store data
  - Create a Vector object in which to store multiple objects.
  - Create a single static PersistentObject.

```
private static Vector _data;
PersistentObject store;
static {
        store = PersistentStore.getPersistentObject( 0xdec6a67096f833cL );
        //key is hash of test.samples.restaurants
        _data = (Vector)store.getContents();
        synchronized (store) {
                if (_data == null) {
                        _data = new Vector();
                        store.setContents(_data);
                        store.commit();
                }
        }
}
```

Manage Custom Objects (Cont.)

- Store data persistently
  - In the class for the objects that you want to store, implement the Persistable interface.

```
private static final class RestaurantInfo implements Persistable {
        private String[] _elements;
        public static final int NAME = 0;
        public static final int ADDRESS = 1;
        public static final int PHONE = 2;
        public static final int SPECIALTY = 3;
        public RestaurantInfo() {
                _elements = new String[4];
                for ( int i = 0; i < _elements.length(); ++i) {
                    _elements[i] = new String("");
                }
        }
        public String getElement(int id) {
            return _elements[id];
        }
        public void setElement(int id, String value) {
            _elements[id] = value;
        }
}
```

Manage Custom Objects (Cont.)

- Save an object
  - Define an object. The following code fragment creates a RestaurantInfo object and uses its set methods to define its components.

```
RestaurantInfo info = new RestaurantInfo();
info.setElement(RestaurantInfo.NAME, namefield.getText());
info.setElement(RestaurantInfo.ADDRESS,addressfield.getText());
info.setElement(RestaurantInfo.PHONE, phonefield.getText());
info.setElement(RestaurantInfo.SPECIALTY, specialtyfield.getText());
```

  - Add the object to a vector by invoking addElement().

```
_data.addElement(info);
```

  - Synchronize with the persistent object so that other threads cannot make changes to the object at the same time.

```
synchronized(store) {
```

  - Invoke setContents().

```
store.setContents(_data);
```

  - To save the updated data, invoke commit() on the PersistentObject.

```
store.commit();
}
```

Manage Custom Objects (Cont.)

- Retrieve the most recently saved object
  - Invoke _data.lastElement().

```
public void run() {
    synchronized(store) {
        _data = (Vector)store.getContents();
        if (!_data.isEmpty()) {
            RestaurantInfo info = (RestaurantInfo)_data.lastElement();
            namefield.setText(info.getElement(RestaurantInfo.NAME));
            addressfield.setText(info.getElement(RestaurantInfo.ADDRESS));
            phonefield.setText(info.getElement(RestaurantInfo.PHONE));
            specialtyfield.setText(info.getElement(
            RestaurantInfo.SPECIALTY));
        }
    }
}
```

Using the MIDP Record Store

- Create a record store
  - Invoke openRecordStore(), and specify true to indicate that the method should create the record store if the record store does not exist.

```
RecordStore store = RecordStore.openRecordStore("Contacts", true);
```

- Add a record
  - Invoke addRecord().

```
int id = store.addRecord(_data.getBytes(), 0, _data.length());
```

- Retrieve a record
  - Invoke getRecord(int, byte[], int), and provide the following parameters:
    - a record ID
    - a byte array
    - an offset

```
byte[] data = new byte[store.getRecordSize(id)];
store.getRecord(id, data, 0);
String dataString = new String(data);
```

Using the MIDP Record Store (Cont.)

- Retrieve all records
  - Invoke openRecordStore().
  - Invoke enumerateRecords() with the following parameters:
    - filter: specifies a RecordFilter object to retrieve a subset of record store records (if null, the method returns all records)
    - comparator: specifies a RecordComparator object to determine the order in which the method returns the records (if null, the method returns the records in any order)
    - keepUpdated: determines if the method keeps the enumeration current with the changes to the record store

```
RecordStore store = RecordStore.openRecordStore("Contacts",
    false);
RecordEnumeration e = store.enumerateRecords(null, null, false);
```

Data Synchronization

- Tools are not provided by RIM to synchronize data to remote data sources
- The synchronization logic must be build into the application
- 4 types of data synchronization

Types of Data Synchronization

- Wireless (BES)
  - The automatic wireless backup process on a BES is designed to back up data from the BlackBerry device to the BlackBerry Enterprise Server.
  - By default, wireless backup is active on the BlackBerry Enterprise Server.
  - When the automatic wireless backup process runs on the BES, the process saves the application data with the user account settings and the other BlackBerry device data that backs up.
- Wireless (XML data)
  - A java application uses XML APIs to generate and parse XML-formatted data to send and receive over a wireless connection.
- Desktop-based (BlackBerry® Desktop Manager Plug-in)
  - An application uses a USB connection to a computer to synchronize data with a desktop BlackBerry Java Application.
  - Requires the following:
    - BlackBerry Desktop Synchronization APIs
    - the BlackBerry Desktop Manager

Types of Data Synchronization (Cont.)

- A desktop BlackBerry Java Application that can read data from the BlackBerry device using the BlackBerry Desktop Manager Plug-Ins adapter.
- A BlackBerry device user must manually start the synchronization process by running the BlackBerry Desktop Manager Plug-in. This notifies the BlackBerry Java Application on the BlackBerry device to send the data to the desktop application.

- Desktop-based (USB protocols)
  - An application uses a USB connection to a computer and native USB protocols to synchronize data with a desktop application.

Backing Up and Restoring Data

- Add support for backing up data over the wireless network
- Access a SyncCollection
- Notify the system when a SyncCollection changes
- Using SyncObjects
- Add support for backing up data with the BlackBerry Desktop Software
- Activate synchronization when the BlackBerry device starts

Support for Backup & Restore Over Wireless Networks

- Setup the BES to back up the application data using automatic wireless backup
  - Implement the OTASyncCapable and CollectionEventSource interfaces.
- Activate the synchronization process when the BlackBerry device starts
  - In the main method, create code that activates the synchronization process.

```
public static void main(String[] args) {
        boolean startup = false;
        for (int i=0; i<args.length; ++i) {
                if (args[i].startsWith("init")) {
                    startup = true;
                }
        }
        if (startup) {
                //enable application for synchronization on startup
                SerialSyncManager.getInstance().enableSynchronization(new
                RestaurantsSync());
        } else {
                RestaurantsSync app = new RestaurantsSync();
                app.enterEventDispatcher();
        }
}
```

- The first time the BlackBerry device starts, the Alternate CLDC Application Entry Point project passes an argument to the application so that the BlackBerry Java Application registers only once.

Support for Backup & Restore Over Wireless Networks (Cont.)

- Create a project that acts as an alternate entry point to the main BlackBerry® Java® Application.
  - MIDlet applications do not support this task.
    - In the BlackBerry® Integrated Development Environment, create a project.
    - Right-click the project, and then click Properties.
    - Click the Application tab.
    - In the Project type drop-down list, click Alternate CLDC Application Entry Point.
    - In the Alternate entry point for drop-down list, click the project that starts the synchronization process.
    - In the Arguments passed to field, type init. Make sure the value you type in the Arguments passed to field matches the value in the startsWith argument in your BlackBerry Java Applications main method.
    - Select the Auto-run on startup option.
    - Select the System module option.
    - Click OK.

Support for Backup & Restore Over Wireless Networks (Cont.)

- Provide a BlackBerry® Java® Application with schema data for a SyncCollection
  - In your implementation of the OTASyncCapable interface, implement the getSchema()method

```
public SyncCollectionSchema getSchema() {// returns our schema
    return _schema;
}
```

- Uniquely identify each record type in a SyncCollection.
  - Invoke the SyncCollectionSchema.setDefaultRecordType()method. The following example shows only one record type, so it uses the default record type:

```
private static final int DEFAULT_RECORD_TYPE = 1;
_schema = new SyncCollectionSchema();
_schema.setDefaultRecordType(DEFAULT_RECORD_TYPE);
```

Support for Backup & Restore Over Wireless Networks (Cont.)

- Uniquely identify each record in a SyncCollection
  - Invoke the SyncCollectionSchema.setKeyFieldIDs() method.

```
private static final int[] KEY_FIELD_IDS =
  new int[] {FIELDTAG_FIRST_NAME,
  FIELDTAG_LAST_NAME};
_schema.setKeyFieldIds(DEFAULT_RECO
  RD_TYPE, KEY_FIELD_IDS);
```

Accessing a SyncCollection

- Retrieve an instance of the SyncCollection from the RunTimeStore.
  - To ensure the BlackBerry® Java® Application works with only one version of the SyncCollection, implement a static method that returns an instance of the SyncCollection.

```
static OTABackupRestoreContactCollection getInstance() {
    RuntimeStore rs = RuntimeStore.getRuntimeStore();
    synchronized( rs ) {
        OTABackupRestoreContactCollection collection = (OTABackupRestoreContactCollection)rs.get( AR_KEY );
        if( collection == null ) {
            collection = new OTABackupRestoreContactCollection();
            rs.put( AR_KEY, collection );
        }
                return collection;
    }
}
```

- Retrieve the SyncCollection from the PersistentStore
  - To provide the application with access to the newest SyncCollection data from the PersistentStore, invoke the PersistentStore.getPersistentObject() method using the ID of the SyncCollection.

```
private PersistentObject _persist; // The persistable object for the contacts.
private Vector _contacts; // The actual contacts.
private static final long PERSISTENT_KEY = 0x266babf899b20b56L;
_persist = PersistentStore.getPersistentObject( PERSISTENT_KEY );
```

Accessing a SyncCollection (Cont.)

- Store the returned data in a vector object.

```
_contacts = (Vector)_persist.getContents();
```

- Create a method to provide the BlackBerry Java Application with the newest SyncCollection data before a wireless data backup session begins.

```
public void beginTransaction() {
    _persist =
      PersistentStore.getPersistentObject(PERSISTENT_KEY);
    _contacts = (Vector)_persist.getContents();
}
```

- Create code to manage the case where the SyncCollection you retrieve from the PersistentStore is empty.

```
if( _contacts == null ) {
    _contacts = new Vector();
    _persist.setContents( _contacts );
    _persist.commit();
}
```

Notify System when SyncCollection Changes

- Use a collection listener to notify the system when a SyncCollection changes.
  - The system invokes CollectionEventSource.addCollectionListener() to create a CollectionListener for each SyncCollection the BlackBerry®Java® Application makes available for wireless backup.
  - Create a private vector object to store the collection of SyncCollection listeners for the BlackBerry Java Application.

```
private Vector _listeners;
_listeners = new CloneableVector();
```

  - Implement the CollectionEventSource.addCollectionListener() method, making sure the method adds a CollectionListener to the vector.

```
public void addCollectionListener(Object listener) {
    _listeners = ListenerUtilities.fastAddListener( _listeners,
    listener );
}
```

Notify System when SyncCollection Changes (Cont.)

- Remove a collection listener.
  - When a CollectionListener is no longer required, the system invokes CollectionEventSource. removeCollectionListener.
  - Implement the CollectionEventSource.removeCollectionListener()method, using the ListenerUtilities.removeListener() method to remove a CollectionListener from the collection of SyncCollection listeners for the BlackBerry® Java® Application.

```
public void removeCollectionListener(Object listener) {
    _listeners = ListenerUtilities.removeListener( _listeners, listener );
}
```

- Notify the system when an element is added to a SyncCollection.
  - Invoke CollectionListener.elementAdded():

```
for( int i=0; i<_listeners.size(); i++ ) {
        CollectionListener cl = (CollectionListener)_listeners.elementAt( i );
        cl.elementAdded( this, object );
}
return true;
}
```

Notify System when SyncCollection Changes (Cont.)

- Notify the system when an element is removed from a SyncCollection
  - Invoke CollectionListener.elementRemoved().
- Notify the system when an element in a SyncCollection is replaced
  - Invoke CollectionListener.elementUpdated().

Using SyncObjects

- Retrieve SyncObjects from the SyncCollection.
  - Implement the getSyncObjects() method.

```
public SyncObject[] getSyncObjects() { //Retrieve the contact data.
        SyncObject[] contactArray = new SyncObject[_contacts.size()];
        for (int i = _contacts.size() - 1; i >= 0; --i) {
            contactArray[i] = (SyncObject)_contacts.elementAt(i);
        }
        return contactArray;
}
```

- Access a specific SyncObject
  - Implement the getSyncObject() method, using the _uid parameter to retrieve a specific SyncObject.

```
public SyncObject getSyncObject(int uid) {
        for (int i = _contacts.size() - 1; i >= 0; --i) {
                SyncObject so = (SyncObject)_contacts.elementAt(i);
                if ( so.getUID() == uid ) return so;
        }
        return null;
}
```

Using SyncObjects (Cont.)

- Add a SyncObject to the SyncCollection
  - Create a method that adds SyncObjects to the PersistentStore object.

```
public boolean addSyncObject(SyncObject object) {
        // Add a contact to the PersistentStore object.
        _contacts.addElement(object);
}
```

- Save a SyncCollection
  - Before a wireless backup session ends, save the newest SyncCollection data.
  - Invoke the setContents() and commit()methods.

```
public void endTransaction() {
        _persist.setContents(_contacts);
        _persist.commit();
}
```

Adding Support for Backing Up Data with Desktop Software

- Let your BlackBerry® Java® Application maintain a collection of synchronized objects, producing and processing valid synchronization data when creating a SyncObject
  - Implement the SyncCollection and SyncConverter interfaces by the same class or by separate classes, depending on the design of the BlackBerry Java Application.
  - Change the main class for the BlackBerry Java Application to implement the SyncCollection and SyncConverter interfaces.

    ```
    public class RestaurantsSync extends UiApplication implements
        RestaurantsSyncResource,SyncCollection, SyncConverter
    ```
- Let persistable objects be synchronization objects
  - Modifiy a class that implements the Persistable interface to implement the SyncObject interface.

    ```
    private static final class RestaurantInfo implements Persistable,
        SyncObject {
    ```
- Create a unique ID for a synchronization object
  - In the persistable class, create an instance variable for storing a unique ID for synchronization operations.

    ```
    private int _uid;
    ```

Adding Support for Backing Up Data with Desktop Software (Cont.)

- Let your main BlackBerry® Java® Application retrieve the unique ID of the synchronization object
  - In the persistable class, implement the getUID() method to return a unique ID for synchronization operations.

```
public int getUID() {
    return _uid;
}
```

- Enable your main BlackBerry® Java® Application to create a synchronization object using a unique ID
  - In the persistable class, create a constructor that accepts a unique ID as a parameter and sets the _uid variable to this value.

```
public RestaurantInfo(int uid) {
        _elements = new String[4];
        for (int i = 0; i < _elements.length; ++i) {
            _elements[i] = "";
        }
        _uid = uid;
}
```

Active Synchronization when the BlackBerry Device Starts

- Activate synchronization when the device starts.
  - The first time the BlackBerry device starts, the Alternate CLDC Application Entry Point project passes an argument to the BlackBerry Java® Application so that the BlackBerry Java Application registers only once.
  - In the main method of the BlackBerry Java Application, create code that activates the synchronization process.

```
public static void main(String[] args) {
        boolean startup = false;
        for (int i=0; i<args.length; ++i) {
                if (args[i].startsWith("init")) {
                    startup = true;
                }
        }
        if (startup) {   //enable the BlackBerry Java Application for
            synchronization on startup
                SerialSyncManager.getInstance().enableSynchronization(new
                RestaurantsSync());
        } else {
                RestaurantsSync app = new RestaurantsSync();
                app.enterEventDispatcher();
        }
}
```

Active Synchronization when the BlackBerry Device Starts (Cont.)

- Create a project that acts as an alternate entry point to the main BlackBerry® Java® Application.
  - If the BlackBerry Java Application is a MIDlet, arguments cannot pass to the BlackBerry Java Application when the BlackBerry device starts.
  - In the BlackBerry® Integrated Development Environment, create a project.
  - Right-click the project, and then click Properties.
  - Click the Application tab.
  - In the Project type drop-down list, click Alternate CLDC Application Entry Point.
  - In the Alternate entry point for drop-down list, click the project that implements synchronization.
  - In the Arguments passed to field, type init. Make sure the value you type in the Arguments passed to field matches the value in the startsWith argument in the main method of the BlackBerry Java Application.
  - Select the Auto-run on startup option.
  - Select the System module option.
  - Click OK.