

Wireless Online Game Development for Mobile Device

Lesson 7

I123-1-A@Peter Lo 2007

1

What have we learnt last week?

- Introduction to MMAPI
- Play Audio files (Wav and Midi format)
- Play MPEG movie from Internet
- Design and play your own Tone Sequence
- Add sound effect to your game



I123-1-A@Peter Lo 2007

2

Generic Connection Framework

- In the MIDP, you interact with the network using the Generic Connection framework.
- The framework is a set of classes and interfaces defined by the CLDC that replaces most of the **java.io** and **java.net** classes defined by J2SE.

I123-1-A@Peter Lo 2007

3

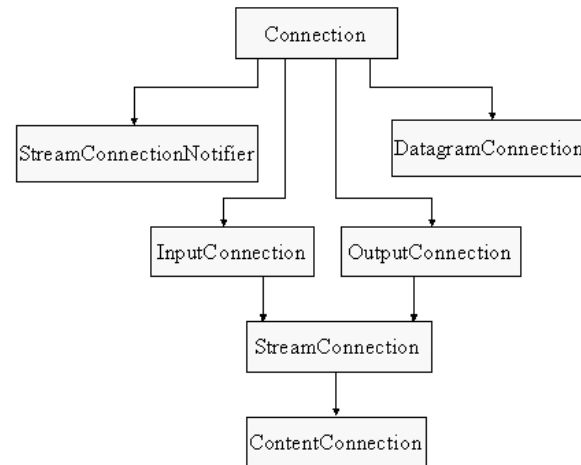
Generic Connection Framework

- In the CLDC Generic Connection framework, all connections are created using the open static method from the Connector class.
- If successful, this method returns an object that implements one of the generic connection interfaces.
- The Connection interface is the base interface such that *StreamConnectionNotifier* is a **Connection** and *InputConnection* is a **Connection** too.

I123-1-A@Peter Lo 2007

4

Connection Interface Hierarchy



I123-1-A@Peter Lo 2007

5

Connection Interface Hierarchy

- The **Connection** interface is the most basic connection type. It can only be opened and closed.
- The **InputConnection** interface represents a device from which data can be read. Its **openInputStream** method returns an input stream for the connection.
- The **OutputConnection** interface represents a device to which data can be written. Its **openOutputStream** method returns an output stream for the connection.
- The **StreamConnection** interface combines the input and output connections.

I123-1-A@Peter Lo 2007

6

Connection Interface Hierarchy

- The **ContentConnection** is a sub-interface of **StreamConnection**. It provides access to some of the basic meta data information provided by HTTP connections.
- The **StreamConnectionNotifier** waits for a connection to be established. It returns a **StreamConnection** on which a communication link has been established.
- The **DatagramConnection** represents a datagram endpoint.

I123-1-A@Peter Lo 2007

7

Package javax.microedition.io

- The **javax.microedition.io** CLDC package contains classes for I/O, including networking I/O.
- To these CLDC classes, the MIDP adds the **HttpConnection** interface for HTTP protocol access.
- This interface defines the necessary methods and constants for an HTTP connection

I123-1-A@Peter Lo 2007

8

Open Connector

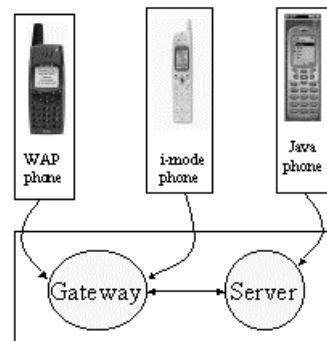
- The open method of the Connector class has the following syntax, where the String parameter has the format "*protocol:address;parameters*".
 - ◆ Connector.open(String);

Connector Examples

- HTTP Connection
 - ◆ Connector.open("http://address");
- Datagram Connection
 - ◆ Connector.open("datagram://address:port#");
- Communicate with a Port
 - ◆ Connector.open("comm:0;baudrate=9600");
- Open Files
 - ◆ Connector.open("file://myFile.txt");

Implementation of HTTP

- The MIDP extends CLDC connectivity to provide support for a subset of the HTTP protocol.
- HTTP can either be implemented using IP protocols (such as TCP/IP) or non-IP protocols (such as WAP and i-mode).



The HttpURLConnection Interface

- The *HttpURLConnection* interface is part of the package *javax.microedition.io*.
- This interface defines the necessary methods and constants for an HTTP connection.
- It has the following signature:
public interface HttpURLConnection
extends javax.microedition.io.ContentConnection

The HttpURLConnection Interface

- The HttpURLConnection interface defines all the usual methods you would expect to see for making HTTP requests and processing the replies included:
 - ◆ setRequestMethod
 - ◆ getHeaderField
 - ◆ openInputStream
- The HttpURLConnection interface makes it simple to interact with any web site on the Internet.

The HTTP Connection

- The HTTP protocol is a request-response application protocol in which the parameters of the request must be set before the request is sent.
- The connection could be in one of the three following states:
 - ◆ **Setup:** No connection yet
 - ◆ **Connected:** Connection has been made, the request has been sent, and some response is expected
 - ◆ **Closed:** Connection is closed

HttpURLConnection Implementation

- In the setup state the following methods can be invoked:
 - ◆ setRequestMethod
 - ◆ setRequestProperty
- For example, suppose you have this connection:
 - ◆ HttpURLConnection conn = (HttpURLConnection) Connector.open ("http://server");
- Then, you can set the request method to be of type POST:
 - ◆ conn.setRequestMethod (HttpURLConnection.POST);
- HTTP properties such as User-Agent can also be set:
 - ◆ c.setRequestProperty("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");

State: Setup → Connected

- If a method requires data to be sent or received from server, there is a state transition from Setup to Connected.
- Examples of methods that cause the transition include:
 - ◆ openInputStream
 - ◆ openOutputStream
 - ◆ openDataInputStream
 - ◆ openDataOutputStream
 - ◆ getLength
 - ◆ getType
 - ◆ getDate
 - ◆ getExpiration

State: Open

- While the connection is open, some of these methods that may be invoked:
 - ◆ `getURL`
 - ◆ `getProtocol`
 - ◆ `getHost`
 - ◆ `getPort`

Make an HTTP request

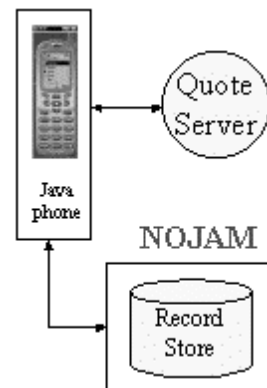
- To make an HTTP request, use the ***Connector.open*** method and a conventional URL, casting the result to an ***HttpConnection***.

```
import javax.microedition.io.*;

HttpConnection conn = Connector.open("http://server");
```

Persistent Storage

- Persistent storage is a non-volatile place for storing the state of objects. Without persistent storage, objects and their states are destroyed when an application closes.
- If you save objects to persistent storage, their lifetime is longer than the program that created them, and later you can read their state and continue to work with them.



Introducing the RMS

- The MIDP provides a mechanism for MIDlets to persistently store data and retrieve it later.
- This mechanism is a simple record-oriented database called the Record Management System (RMS).
- A MIDP database (or a record store) consists of a collection of records that remain persistent after the MIDlet exits.
- When you invoke the MIDlet again, it can retrieve data from the persistent record store.

Introducing the Record Store

- Record stores are platform-dependent because they are created in platform-dependent locations.
- MIDlets within a single application can create multiple record stores with different names.
- The RMS APIs provide the following functionality:
 - ◆ Allow MIDlets to manipulate records within a record store.
 - ◆ Allow MIDlets in the same application to share records.

Working with Threads

- The MIDP RMS implementation ensures that all individual record store operations are atomic, synchronous, and serialized, so no corruption occurs with multiple access.
- If your MIDlets use multiple threads to access a record store, it is your responsibility to synchronize this access, or some of your records might be overwritten.

The RMS Package

- To use the RMS, import the *javax.microedition.rms* package.
- The RMS package consists of the following four interfaces, one class, and five exception classes:

Interfaces

- RecordComparator:
 - ◆ Defines a comparator to compare two records.
- RecordEnumeration:
 - ◆ Represents a bidirectional record enumerator.
- RecordFilter:
 - ◆ Defines a filter to examine a record and checks if it matches based on a criteria defined by the application.
- RecordListener:
 - ◆ Receives records which were added, changed, or deleted from a record store.

Classes

- RecordStore:
 - ◆ Represents a record store.

Exceptions

- InvalidRecordIDException:
 - ◆ Thrown to indicate the RecordID is invalid.
- RecordStoreException:
 - ◆ Thrown to indicate a general exception was thrown.
- RecordStoreFullException:
 - ◆ Thrown to indicate the record store file system is full.
- RecordStoreNotFoundException:
 - ◆ Thrown to indicate the record store could not be found.
- RecordStoreNotOpenException:
 - ◆ Thrown to indicate an operation on a closed record store.

What is a Record Store?

- A record store consists of a collection of records that are uniquely identified by their record ID, which is an integer value.
- The record ID is the primary key for the records. The first record has an ID of 1, and each additional record is assigned an ID that is the previous value plus 1.

Opening a Record Store

- To open a record store, use the *openRecordStore()* static method:
 - ◆ *RecordStore db = RecordStore.openRecordStore ("myDBfile", true);*
- This code creates a new database file named “myDBfile”.
- The second parameter, which is set to true, says that if the record store does not exist, create it.

Creating a New Record

- A record is an array of bytes.
- You can use the *DataInputStream*, *DataOutputStream*, *ByteArrayInputStream* and *ByteArrayOutputStream* classes to pack and unpack data types into and out of the byte arrays.
- The first record created has an ID of 1 and is the primary key. The second record has the previous ID + 1, ...etc.
- You construct a *DataOutputStream* for writing the record to the record store, then you convert the *ByteArrayOutputStream* to a byte array, and finally you invoke *addRecord()* to add the record to the record store.

Example

- Suppose you have the following string record: FirstName, LastName, Age. To add this record to the record store, use the *addRecord()* method as follows:
 - ◆ `ByteArrayOutputStream baos = new ByteArrayOutputStream();`
 - ◆ `DataOutputStream dos = new DataOutputStream(baos);`
 - ◆ `dos.writeUTF(record);`
 - ◆ `byte[] b = baos.toByteArray();`
 - ◆ `db.addRecord(b, 0, b.length);`

Reading Data from the Record Store

- To read a record from the record store, you construct input streams instead of output streams. This is done as follows:
 - ◆ `ByteArrayInputStream bais = new ByteArrayInputStream(record1);`
 - ◆ `DataInputStream dis = new DataInputStream(bais);`
 - ◆ `String in = dis.readUTF();`

Deleting a Record from the Record Store

- To delete the record, you have to know the record ID for the record to be deleted. Then use the *deleteRecord()* method. This method takes an integer as a parameter, which is the record ID of the record to be deleted.
- There is no method to get the record ID. To work around this, every time you create a new record, add its record ID to a vector like this:
 - ◆ `Vector recordIDs = new Vector();`
 - ◆ `int lastID = 1;`
 - ◆ `db.addRecord();`
 - ◆ `recordIDs.addElement(new Integer(++lastID));`

Deleting a Record from the Record Store

- To delete a record, find the record ID of the record you want to delete:
- Compare to see if this is the record you want by invoking *compare()* which is shown next. Then call *db.deleteRecord(id)*;
 - ◆ Enumeration IDs = recordIDs.elements();
 - ◆ while(IDs.hasMoreElements()) {
 - ◆ int id = ((Integer) IDs.nextElement()).intValue();
 - ◆ }

Comparing my Record with Records in the Record Store

- To search for the right record to delete, your application must implement the Comparator interface (by providing an implementation to the compare method) to compare two records.
- The return value indicates the ordering of the two records.

Example

- Suppose you want to compare two strings that you retrieved from two records.
 - ◆ public someClas implements Comparator {
 - ◆ public int compare(byte[] record1, byte[] record2) {
 - ByteArrayInputStream bais1 = new ByteArrayInputStream(record1);
 - DataInputStream dis1 = new DataInputStream(bais1);
 - ByteArrayInputStream bais2 = new ByteArrayInputStream(record2);
 - DataInputStream dis2 = new DataInputStream(bais2);
 - String name1 = dis1.readUTF();
 - String name2 = dis2.readUTF();
 - int num = name1.compareTo(name2);
 - if (num > 0) { return RecordComparator.FOLLOWS;
 - } else if (num < 0) { return recordcomparator.precedes;
 - } else {return recordcomparator.equivalent;
 - }
 - ◆ }

Record Comparator Constants

- The constants *FOLLOWS*, *PRECEDES*, and *EQUIVALENT* are defined in the *RecordComparator* interface and have the following meanings:
 - ◆ *FOLLOWS*: Its value is 1 and means the left parameter follows the right parameter in terms of search or sort order. ($A > B$)
 - ◆ *PRECEDES*: Its value is -1 and means the left parameter precedes the right parameter in terms on search or sort order. ($A < B$)
 - ◆ *EQUIVALENT*: Its value is 0 and means the two parameters are the same. ($A = B$)

Closing the Record Store

- To close the record store, use the *closeRecordStore()* method

Making a Good Game

- There are 5 steps to creating a game, from the moment the concept enters the space between your ears to the moment your computer screams "File's Done".
 - Conceptualization
 - Design
 - Programming
 - Testing
 - Release and Marketing

Conceptualization

- This is the part where a game idea enters your head. For whatever reason, you were thinking about something, and suddenly boom, your brain is fixated on this thought.
- Take the game concept and "play the game" inside your head – imagine how it would appear on a computer screen. Our brains are weird, so you shall have a tough time recreating exactly what your mind sees when it comes to it.
- Ask yourself while you're "playing", "Does this game seem fun? Is it something I could get into? Is it something other people will enjoy?"
- *If you answered **NO** to any of those questions, you may want to reevaluate your concept.*

Plan your Entire Game

- You should plan out every little part of the game.
- Need to get the plot and storyline down.
- Write down each of the events that is going to happen in order.
- Write down what all the main characters are going to say.
- Write down what location each event takes place in.

Story Design

- The story is the driving force behind any RPG.
- All of your characters and events must be told believably and expressed through your story. Without one, your RPG will lose a lot of it's appeal.
- Why is the story so important?
 - ◆ It is one of the main things that keeps your players interested.
 - ◆ If the only thing keeping your game going is the graphics and game play, the player is likely to get bored and enjoy the game much less.

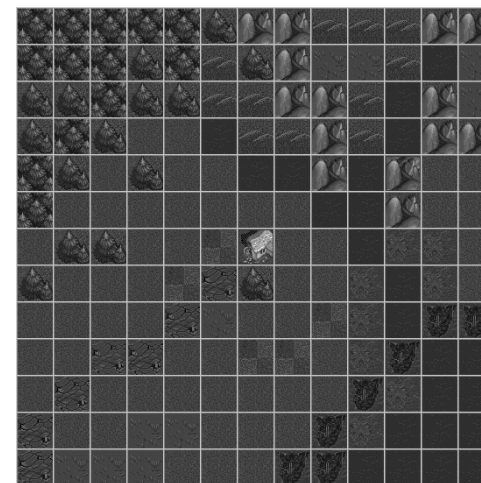
How to write a good story?







- It makes things much easier if you base a lot of the events on things that happened in your real life. You already know the experiences and it makes it easy to write it out.
- Look to other great stories for inspiration. Don't steal their stories, try to do something similar in your game. Only make it original by adding your own twist to it.
- Tether the story down to reality as you did with the characters. If the player feels as though it could really happen, he or she will probably enjoy it a lot more.
- To summarize, make the story believable and base it on experiences of your own life. It's one of the most important parts of your RPG, so give it the proper attention.

Map Design

- Working out what the world map and locations: Give your world a structure, and figure out where there will be towns and dungeons and everything else.
- Draw a little square for each card and put it's number inside, then connect the squares with lines to show where you can move to inside.
- It should come out so that you know the basic layout of each location and have an idea of what card will go where.

Simple Map



-  Village
-  Mountain
-  Marsh
-  Water
-  Forest
-  Field

Movement Design

- During the game you will spend much of your time moving from one place to another. There are several different methods of movement depending on your current location.
- The three main types of movement are Town Movement, Wilderness Movement, and Dungeon Movement.
- Each type should have its own rules and effects.

Character Design

- Every game has characters in it, but in RPG the characters are especially important.
- The characters have to be believable and must be able to get the player emotionally attached to them.
- You must weave a good story around them and let it unfold, with the player right there with the character every step of the way.
- If you do, the story will have a lot more meaning and contribute to the game much more.

What kind of characters should you have?

- You may have characters who are shy, who are confident, follow a religion devoutly, or anything else you want. Make their personalities believable, make the player care for the character.

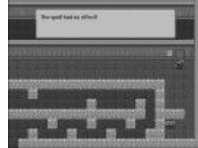


How to Design?

- Base on the experiences from your own life.
 - ◆ You know best how you felt when something happened to you, you know what different types of people are like.
 - ◆ Make characters that other people can relate to.
- Make your characters unique.
 - ◆ While you will have to have a lot of similarities to other video game characters, part of being human is being unique, and if your character is not in the least unique then they will be less believable.



Dungeon Design



- Don't make the whole map as a dungeons. Dungeon usually in a castle or similar place.
- Make the places as unique and original as possible. Try to keep your player interested.
- Going through one after another of the same type of place get old real fast. Try to keep things fresh.
- Don't make your places with confusing, almost completely random layout.
- Add puzzles in appropriate areas. You'll really have to use your own judgment here, so just try to be original with it.

What about the enemies in your dungeons?



- This one relies heavily on common sense.
 - ◆ In a forest, you might have evil trees that attack you.
 - ◆ In a castle dungeon, you might be fighting off the guards so you can escape.
 - ◆ However, a prison guard would not be running around a forest, nor would a evil tree be in some castle dungeon.
- Use logic and try to make the enemies interesting. Don't make the majority of the enemies the exact same thing, but with different coloring. That brings down the quality of the game quite a bit.

