Developing a 2D game in Java ME - Part 5

This article shows how to implement a high score table for the Java ME game.

This is the fifth in a series of articles that cover all the basics of developing an application for mobile devices using Java ME, learning the main libraries, classes, and methods available in Java ME.

24 Aug 2008

Introduction

After the last lesson, the Arkanoid MIDlet was completely functional but saving high scores had not yet been implemented. The high scores were lost each time the application was closed, and the same happened for the game settings.

To implement this functionality you need to understand how the MIDlet IO works and how to use record stores.

Streams

Let's start with simple IO operations. The classes that implement them are in the java.io package. Java ME has only a fraction of the classes available in Java SE but they are the most useful ones:

- **InputStream, OutputStream**: the base classes for binary streams
- **ByteArrayInputStream, ByteArrayOutputStream**: streams to buffer arrays in the memory
- **DataInputStream, DataOutputStream**: read and write primitive Java types (int, float, ...) to streams
- **Reader, Writer**: the base classes for character streams
- **OutputStreamWriter, InputStreamReader**: classes to read character streams using different encodings

To learn usage of these classes, create a settings file that store some game options:

- Number of lifes
- Ball speed
- Time to complete a level
- Number of points for each brick hit

To implement this settings file, create a simple text file `settings.txt` where each line represents a setting. The setting name and its value are separated by a "="-character.

```
ball_speed=2
start_lifes=4
level_time=100
brick_points=10
```

Add the file to your project resources and make sure this file is stored inside the JAR file with the classes. To access the file, you need use the method `getResourceAsStream(String name)` from the Class class. This method gives you access to any file stored inside the JAR file. The path "/" points to the root level of the JAR file. For example, the following code can be used to access the settings file:

```
InputStream is = this.getClass().getResourceAsStream("/settings.txt");
// read the first byte
byte c = is.read();
```

This code returns the first byte of the file, but what you really need is to read each line as a character stream. Use the InputStreamReader for this and then parse each line for the setting's key and value.

```
public class Settings {
  public Hashtable values;

  public Settings(String file) {
    values = new Hashtable(10);
    read(file);
  }

  public String getSetting(String key){
    return (String)(values.get(key));
  }

  /**
   * Opens the file and reads all the settings to the hashtable
   * @param file, the name of the file to read
   */
  public void read(String file){
    // open file
    InputStream is = this.getClass().getResourceAsStream(file);
    InputStreamReader isr = new InputStreamReader(is);
    // create a buffer to store lines
    StringBuffer lineBuffer = new StringBuffer();
    int c;
    try {
      c = isr.read();
      while(c != -1){
        lineBuffer.append((char)c);
        c = isr.read();
        // checks for end of line character
        if ( c == 10 || c==-1){
          // cleans extra spaces or end of lines chars
          String line = lineBuffer.toString().trim();
          // splits the string using the = character
          int pos = line.indexOf("=");
```


(C) Copyright Nokia 2013. All rights reserved.

```
          if (pos != -1){
            // adds a new setting
            String key = line.substring(0,pos);
            String value = line.substring(pos+1);
            values.put(key, value);
          }
          // clean buffer
          lineBuffer.setLength(0);
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      try {
        is.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Now just use this class in the `init()` method in the Canvas class:

```
 public void initSettings() {
    Settings setting = new Settings("/settings.txt");
    BALL_SPEED = Integer.parseInt(setting.getSetting("ball_speed"));
    MAX_LIFES = Integer.parseInt(setting.getSetting("start_lifes"));
    MAX_TIME = Integer.parseInt(setting.getSetting("level_time"));
    BRICK_SCORE = Integer.parseInt(setting.getSetting("brick_points"));
}

public void init() {
  initSettings();
  [...]
}
```

The configuration file has now been created. However, the `getResourceAsStream()` method only allows reading files stored in the JAR file but not writing to them.

RecordStore

To write data persistently you need to use the RecordStore class. This class allows creating small databases where you can store and retrieve data that is persistent between MIDlet launches.

This class is implemented in the javax.microedition.rms package and it provides the following static constructor methods:

- openRecordStore(String recordStoreName, boolean createIfNecessary)
- openRecordStore(String recordStoreName, boolean createIfNecessary, int authmode, boolean writable)
- openRecordStore(String recordStoreName, String vendorName, String suiteName)

These methods allow creating and/or opening a record store. The names of the record stores are case sensitive and have a maximum length of 32 characters. One special feature that needs to be taken into account is the authmode setting. This setting has two values:

- **AUTHMODE_PRIVATE**: In this mode only the MIDlet that created the RecordStore has access to it.
- **AUTHMODE_ANY**: In this mode any MIDlet can open the record store, read data from it, and if the `writable` setting is `true`, write data to it. This allows sharing data between MIDlets, so you can, for example, have two games that share the high-score tables. RecordStores are uniquely identified by their creation name and also by the MIDlet-Name and MIDlet-Vendor properties from the JAD file.

After you open the RecordStore, you can add records to it. Each record is made of an array of bytes and it gets a unique ID when it is created. After you create a record you can retrieve its data, change the data, and delete it. The following methods are available:

- addRecord(byte[] data, int offset, int numBytes)
- deleteRecord(int recordId)
- setRecord(int recordId, byte[] newData, int offset, int numBytes)

The maximum size of a RecordStore depends on the device. You can use the `getSizeAvailable()` method, but the returned value is not always accurate.

Use the RecordStore class to save high scores:

```
public void saveData() {
  try {
    // open recordstore options
    RecordStore options = RecordStore.openRecordStore("options", true);
    byte[] data = saveOptions();
    // check that record store is not empty
    if (options.getNumRecords() != 0) {
      // update the settings
      options.setRecord(1, data, 0, data.length);
    } else {
      // adds the settings
      options.addRecord(data, 0, data.length);
    }
    // closes the record store
    options.closeRecordStore();
  } catch (RecordStoreException ex) {
  }
}

public byte[] saveOptions() {
  // create a byte array stream to store data temporarily
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  DataOutputStream dos = new DataOutputStream(baos);
  try {
    dos.writeBoolean(soundOn);
    // write scores
    for (int i = 0; i < scores.length; i++) {
      dos.writeInt(scores[i].value);
      dos.writeUTF(scores[i].name);
      dos.writeLong(scores[i].when.getTime());
```


(C) Copyright Nokia 2013. All rights reserved.

```
    }
    // push all the data to the byte array stream
    dos.flush();
  } catch (IOException ex) {
  }
  // returns bytes from stream
  return baos.toByteArray();
}
```

This "options" record store is used to store the settings and the high scores. Data is written to a `DataOutputStream` and a `ByteArrayOutputStream` is used to convert data to a binary array.

Now implement reading of the data:

```
public void loadData() {
  try {
    RecordStore options = RecordStore.openRecordStore("options", true);
    // check that record store is not empty
    if (options.getNumRecords() != 0) {
      loadOptions(options.getRecord(1));
    }
    options.closeRecordStore();
  } catch (RecordStoreException ex) {
  }
}

public void loadOptions(byte[] data) {
  // create a byte array stream to store data temporarily
  ByteArrayInputStream bais = new ByteArrayInputStream(data);
  // creates a data input stream to read from
  DataInputStream dis = new DataInputStream(bais);
  try {
    soundOn = dis.readBoolean();
    // read scores
    for (int i = 0; i < scores.length; i++) {
      int value = dis.readInt();
      String name = dis.readUTF();
      Date date = new Date(dis.readLong());
      scores[i] = new Score(value, name, date);
    }
    dis.close();
  } catch (IOException ex) {
  }
}
```

Simply open the record store and check if the record is available. If true, retrieve the data and use `DataInputStream` combined with `ByteArrayInputStream`.

Now add calls for these methods to the startup and shutdown of the MIDlet.

```
public void initOptions() {
  soundOn = true;

  if (scores == null) {
    scores = new Score[10];
    for (int i = 0; i < scores.length; i++) {
      scores[i] = new Score(0, "Empty", new Date());
    }
  }
  // loads data from recordstore if available
  loadData();
}

public void exit() {
  // store high scores and settings to recordstore
  saveData();
  notifyDestroyed();
}
```

After these changes, the game saves the player's high scores and settings. The next article discusses using the game settings and adding sound to the game.

Downloads

- Source code
- Binaries

Go to Developing a 2D game in Java ME - Part 6

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.