

Over the past few years, concerns about security on the Internet have heated up immensely. It's common in this day and age to hear of companies whose data has been hacked and from whom valuable credit card information has been stolen. It probably won't be long before people can "sniff" the data going through cell phones just as easily as data traveling across the Internet. What can you do? Fortunately, computers these days are capable of using *encryption* techniques to scramble data as it travels through the unsafe corridors of the Internet. In fact, one of the most popular forms of encryption uses a wide variety of cryptographic techniques to protect your data. It's called Secure Sockets Layer (SSL) and is built into practically every web browser.

Will you need encryption for your MIDlet programs? That's a hard question to answer. Many wireless protocols already use a sophisticated level of scrambling—far more than an average "sniffer" can decode. And while cryptographic software can be relatively small and easy to use on a desktop, using it on a cell phone can quickly expend both your processing power and program space. Understanding cryptography can take a bit of time as well.^[2] However, if you absolutely must have security on your cell phone to protect data traveling on the Internet, we recommend checking out the open source lightweight API software from "The Legion of the Bouncy Castle" (<http://www.bouncycastle.org>).

^[2] A great place to start, however, is Bruce Schneier's *Applied Cryptography*, Second Edition (Wiley).

Chapter 8. Database Programming

A database is a non-volatile place for storing the state of objects. For some applications, you might need objects to exist even after the application that created them closes. Without a storage mechanism, objects and their states are destroyed when an application closes. However, if you save objects to a *persistent storage facility*, such as a database, they can be read in later by the same application or even other applications.

The persistent storage facilities provided in the J2SE platform, such as the JDBC and Object Serialization APIs, are too large for handheld devices with a small memory footprint. Storage requirements vary significantly from one resource-constrained device to another. J2ME and the MIDP solve this problem by using the Record Management System (RMS).

This chapter introduces the details of the MIDP RMS, a persistent storage facility for MIDlets, and shows you how to develop MIDP database applications, using an example involving stocks. Throughout this chapter, the terms *record store* and *database* are used interchangeably.

8.1 The Record Management System

An RMS database (or record store) consists of a collection of records that remain persistent after the MIDlet closes. When you invoke the MIDlet again, it can retrieve data from the persistent record store. However, to use the RMS, we need to get familiar with some of the classes and concepts provided by the `javax.microedition.rms` package.

8.1.1 Record Stores

Record stores, which are binary files, are actually platform-dependent because they are created in platform-unique locations. MIDlets within a single application (a MIDlet suite) can create multiple record stores with different names. The RMS APIs provide the following types of functionality.

- They allow MIDlets to manipulate (add and remove) records within a record store.
- They allow MIDlets in the same application or suite to share records (access each other's record store directly).

Note that no mechanism, however, is provided for sharing records between MIDlets in different MIDlet suites.

Record stores have names that are case-sensitive and cannot be more than 32 characters in length. A MIDlet cannot create two record stores with the same name in the same application. However, it can create a record store with the same name in another application. When you create a new record store in an emulator, it is typically stored under a directory called *NOJAM*. For example, let's assume you are using the Wireless Toolkit and it is installed under *C:\J2MEWTK*. If your project name is *StockQuotes* and your record store is *mystocks*, the record store is created under *C:\J2MEWTK\NOJAM* and has the name *mystocks.db*.

The MIDP RMS implementation ensures that all individual record store operations are atomic, synchronous, and serialized, so no corruption occurs with multiple access. However, if your MIDlets use multiple threads to access a record store, it is your responsibility to synchronize this access. Otherwise, some of your records might accidentally be overwritten if your application is not thread-safe.

8.1.2 The `javax.microedition.rms` Package

The `javax.microedition.rms` package consists of four interfaces, one class, and five exception classes.

8.1.2.1 Interfaces

[Table 8-1](#) lists the four interfaces in the `javax.microedition.rms` package.

Table 8-1. The interfaces in <code>javax.microedition.rms</code>	
Interface	Description
<code>RecordComparator</code>	This interface defines a comparator to compare two records.
<code>RecordEnumeration</code>	This interface represents a bidirectional record enumerator.
<code>RecordFilter</code>	This interface defines a filter to examine a record and check if it matches, based on a criteria defined by the application.
<code>RecordListener</code>	This interface receives records that were added, changed, or deleted from a record store.

8.1.2.2 Classes

There is one class in this package, as shown in [Table 8-2](#).

Table 8-2. The class in javax.microedition.rms	
Class	Description
RecordStore	This class represents a record store.

8.1.2.3 Exceptions

There are five exceptions in the package, as shown in [Table 8-3](#).

Table 8-3. The exceptions in javax.microedition.rms	
Exception	Description
InvalidRecordIDException	This exception is thrown to indicate that the RecordID is invalid.
RecordStoreException	This exception is thrown when a general exception is thrown by the RecordStore class.
RecordStoreFullException	This exception is thrown when the record store filesystem is full.
RecordStoreNotFoundException	This exception is thrown when the record store could not be found.
RecordStoreNotOpenException	This exception is thrown to indicate an operation on a closed record store.

8.2 Programming with the RMS

Database programming with the RMS is relatively straightforward. A record store consists of a collection of records that is uniquely identified by its 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. The record ID is stored as an integer value, which gives the theoretical limit of 2,147,483,647 records.^[1]

^[1] But if your devices had at least 2.1 gig of memory, you probably wouldn't need to use the J2ME!

8.2.1 Opening, Closing, and Deleting a Record Store

To open a record store, you need to be familiar with the static `openRecordStore( )` method of the `RecordStore` class:

```
public static RecordStore openRecordStore(String recordStoreName,
    Boolean createIfNecessary) throws RecordStoreException,
    RecordStoreFullException, RecordStoreNotFoundException
```

Here is an example of using this method:

```
RecordStore db = null;

try {
    db = RecordStore.openRecordStore("myDBfile", true);
} catch (RecordStoreNotFoundException rsnfe) {
    // Handle exception
} catch (RecordStoreFullException fsfe) {
    // Handle exception
} catch (RecordStoreException rse) {
    // Handle exception
```

```
}
```

Assuming that everything works right, this line of code creates a new database file named *myDBfile*. The second parameter, a boolean which is set to `true`, says that if the record store does not exist, then you should create it.



If the `openRecordStore( )` method is called by a MIDlet when the record store is already open by another MIDlet in the same MIDlet suite, the method returns a reference to the same `RecordStore` object.

Once we've opened a record store, we will eventually need to close it. We can do this with the following `RecordStore` method:

```
public void closeRecordStore( ) throws RecordStoreNotOpenException,  
    RecordStoreException
```

It is important to note that the record store will not actually be closed until `closeRecordStore()` is called as many times as `openRecordStore( )` was called. Therefore, the programmer must balance the number of close calls and open calls before the record store is actually closed. Keeping a record store open can take up a great deal of memory. Consider closing a record store even when a MIDlet is placed in the paused state.

Sometimes it's necessary to locate a particular record store among several that are currently on the device. If you want to find out the names of all the record stores currently on the device, use the following static method:

```
public static String[] listRecordStores( )
```

You can delete an entire record store from the database, using the following static `RecordStore` method:

```
public static void deleteRecordStore(String recordStoreName) throws  
    RecordStoreException, RecordStoreNotFoundException
```

You can find out the size of the currently opened record store in bytes, using the `getSize( )` method:

```
public int getSize( ) throws RecordStoreNotOpenException
```

In addition, if you want to find out how many bytes the current record store can still grow, use the following:

```
public int getSizeAvailable( ) throws RecordStoreNotOpenException
```

Finally, you can use the `getVersion( )` method of the `RecordStore` to find out the "version" of the current record store. Here, the version does not have anything to do with the version of software that the database is using. Instead, the version is actually

an integer stored with the record store that increments each time a record is added, modified, or deleted.

```
public int getVersion( ) throws RecordStoreNotOpenException
```

8.2.2 Creating and Modifying Records

A record is simply an array of bytes. You can use the `DataInputStream`, `DataOutputStream`, `ByteArrayInputStream`, and `ByteArrayOutputStream` classes to pack and unpack data in and out of byte arrays. For example, suppose you have the following record, represented by a single string: "Firstname, LastName, Age". To add this record to the record store, you can use the `addRecord( )` method:

```
public int addRecord(byte[] data, int offset, int numBytes) throws
    RecordStoreNotOpenException, RecordStoreException,
    RecordStoreFullException
```

This method adds all or part of the contents of a byte array of data, starting at the `offset` specified and continuing the `numBytes`, and places it in the record store. The method then returns the index assigned to that record in the database. Continuing from the previous example:

```
try {
    ByteArrayOutputStream baos = new ByteArrayOutputStream( );
    DataOutputStream dos = new DataOutputStream(baos);
    dos.writeUTF(record);
    Byte b[] = baos.toByteArray( );
    recordNumber = db.addRecord(b, 0, b.length);
} catch (Exception e) {
    // Handle exceptions
}
```

Here, we construct a `DataOutputStream` for writing the record to the record store, then convert the `ByteArrayOutputStream` to a byte array. Finally, we invoke `addRecord( )` to add the record to the record store. This is not the only way to construct and add a new record to a record store, however. Instead of creating a series of linked streams, it is easier to convert a string into a series of bytes using the `getBytes( )` method of `String`:

```
try {
    String record = "Firstname, Lastname, Age";
    Byte b[] = record.getBytes( );
    recordNumber = db.addRecord(b, 0, b.length);
} catch (Exception e) {
    // Handle Exceptions
}
```

You can also use the `setRecord( )` method if you want to explicitly reset an indexed record in the record store. This works the same as `addRecord( )`, except that the first parameter is the specific record ID that you wish to set:

```
public void setRecord(int recordId, byte[] newData, int offset,
    int numBytes) throws RecordStoreNotOpenException,
    InvalidRecordIDException, RecordStoreException,
```

RecordStoreFullException

To read a record from the record store, you can use one of two `getRecord( )` methods:

```
public byte[] getRecord(int recordID) throws
    RecordStoreNotOpenException, InvalidRecordIDException,
    RecordStoreException
public int getRecord(int recordID, byte[] buffer, int offset)
    throws RecordStoreNotOpenException, InvalidRecordIDException,
    RecordStoreException
```

The first method returns a byte array containing the entire record that was stored at the specified record ID. The second method will attempt to fill the byte array passed in starting at the specified offset with the contents of the specified record ID. The second method will return the amount of bytes actually copied as an integer. Be sure that there is enough room in the byte array to handle the data from the record, or an exception will be thrown.

To extract the data from a byte array, we can do the opposite of what we did before: construct input streams instead of output streams. Here is an example:

```
String in = null;

try {
    byte[] record = new byte[db.getRecordSize(recordNumber)];
    db.getRecord(recordNumber, record, 0);
    ByteArrayInputStream bais = new ByteArrayInputStream(record);
    DataInputStream dis = new DataInputStream(bais);
    in = dis.readUTF( );
} catch (Exception e) {
    // Handle exceptions
}
```

Or, if we don't want to filter the data through any sort of stream, we can take the easy route and simply pass the byte array directly into the `String` constructor:

```
String in = null;

try {
    byte[] record = new byte[db.getRecordSize(recordNumber)];
    db.getRecord(recordNumber, record, 0);
    in = new String(record);
} catch (Exception e) {
    // Handle exceptions
}
```

To delete a record from the record store, you have to know the record ID of the record to be deleted. To delete the record, use the `deleteRecord( )` method.

```
public void deleteRecord(int recordID) throws
    RecordStoreNotOpenException, InvalidRecordIDException,
    RecordStoreException
```

Note that the other records will not change their ID. In fact, the record store will not reuse the ID of a record once it is deleted.

There are a number of other methods that you can use in the `RecordStore` class. If you want to find out when the record store was last modified, you can use the following method:

```
public long getLastModified( ) throws RecordStoreNotOpenException
```

This method returns a date in the form of a `long`, which is equivalent to the format used by `System.currentTimeMillis( )`. This can be passed into the constructor of the `java.util.Date` object, which can in turn be used by the `java.util.Calendar` object, as well as the `javax.microedition.lcdui.DateField` component.

If you want to know the name of the current record store, use the `getName( )` method:

```
public String getName( ) throws RecordStoreNotOpenException
```

If you want to find out the next ID that the database will use when storing a record, you can use the `getNextRecordID( )` method:

```
public int getNextRecordID( ) throws RecordStoreNotOpenException,  
    RecordStoreException
```

To get a tally of the number of records currently in the record store, use the following method:

```
public int getNumRecords( ) throws RecordStoreNotOpenException
```

If you wish to find out the number of bytes in a currently stored record, use the `getRecordSize( )` method (as we did in one of the previous examples to initialize the receiving byte array):

```
public int getRecordSize(int recordId) throws  
    RecordStoreNotOpenException, InvalidRecordIDException,  
    RecordStoreException
```

Finally, there is one other method that `RecordStore` includes that allows us to enumerate all the records located in the current record store. It looks like the following:

```
public RecordEnumeration enumerateRecords(RecordFilter filter,  
    RecordComparator comparator, boolean keepUpdated) throws  
    RecordStoreNotOpenException
```

This method will list all the records in the record store, first using the appropriate filter to select those records, then sorting them using the record comparator, and finally returning the results inside a specially designed enumeration object. However, there are several interfaces that we need to go over first before we can grasp the broad picture of what this method does.

8.2.3 Filtering Records

First, the `enumerateRecords()` method must determine which records will be included in the enumeration that you are requesting. The method determines this by passing in an object that implements the `javax.microedition.rms.RecordFilter` interface. Luckily, only one method in this interface needs to be implemented:

```
public boolean matches(byte[] candidate);
```

This method takes in a byte array that represents a candidate record from a record store. The implementation must return a boolean that indicates whether the record should be included in the enumeration. If the method returns `true`, the record will be included; if it's `false`, it will not be. If you want all records to be included in the enumeration, simply pass in `null` to that parameter of the `enumerateRecords()` method.

Here is a sample implementation for the `RecordFilter` interface that only accepts String-based records that start with the letters "JULY":

```
import javax.microedition.rms.*;

public class MyFilter implements RecordFilter {

    public boolean matches(byte[] candidate) {
        String c = new String(candidate);
        if (c.startsWith("JULY"))
            return true;
        else
            return false;
    }
}
```

8.2.4 Comparing Records

The `enumerateRecords()` method has the ability to sort the records that it is returning. If you would like it to do so, you must give it the ability to compare records in the record store. For this, your application must provide an object that implements the `javax.microedition.rms.RecordComparator` interface. Again, this is relatively simple, as you must implement only one method in this interface:

```
public int compare(byte[] record1, byte[] record2)
```

The return value of this method indicates how the two records compare, and it can be one of three constants. For example, suppose you want to lexicographically compare two strings that you retrieved from two records. Here is a sample implementation:

```
import javax.microedition.rms.*;

public class MyComparator implements RecordComparator {

    public int compare(byte record1[], byte record2[]) {

        String name1 = new String(record1);
        String name2 = new String(record2);
```

```

        int num = name1.compareTo(name2);

        if(num > 0) {
            return RecordComparator.FOLLOWS;
        } else if (num < 0) {
            return RecordComparator.PRECEDES;
        } else {
            return RecordComparator.EQUIVALENT;
        }
    }
}

```

The constants that the object should return, `RecordComparator.FOLLOWS`, `RecordComparator.PRECEDES`, and `RecordComparator.EQUIVALENT`, are declared in the `RecordComparator` interface and have the following meanings:

`RecordComparator.FOLLOWS`

The first record follows the second record in terms of search or sort order.

`RecordComparator.PRECEDES`

The first record precedes the second record in terms of search or sort order.

`RecordComparator.EQUIVALENT`

The two records are the same.

If the ID of each of the records is acceptable as an order, then you can pass in `null` to that parameter of the `enumerateRecords( )` method.

8.2.5 Enumerating Records

Finally, the `enumerateRecords( )` method will return to you an object that implements the `javax.microedition.rms.RecordEnumeration` interface. This interface is used to provide a standard set of methods to access the enumeration, and acts like a more sophisticated version of the `java.util.Enumeration` class. It looks like the following:

```

public interface RecordEnumeration {

    public void destroy( );
    public boolean hasNextElement( );
    public boolean hasPreviousElement( );
    public boolean isKeptUpdated( );
    public void keepUpdated(boolean keepUpdated);
    public byte[] nextRecord( ) throws InvalidRecordIDException,
        RecordStoreNotOpenException, RecordStoreException;
    public int nextRecordId( ) throws InvalidRecordIDException;
    public int numRecords( );
    public byte[] previousRecord( ) throws InvalidRecordIDException,
        RecordStoreNotOpenException, RecordStoreException;
    public int previousRecordID( ) throws InvalidRecordIDException;
    public void rebuild( );
}

```

```
public void reset( );  
  
}
```

If you recall the signature of the `enumerateRecords( )` method, you'll remember that it had a third parameter, which was a boolean, called `keepUpdated`. The function of this boolean is to have the enumeration monitor the current record store. If there are several threads updating the record store at any given time, the enumeration may not have the correct values inside it.

However, if the boolean parameter to `enumerateRecords( )` is set to `true`, the `RecordEnumeration` becomes a listener to the record store. If there are any changes to the records, the `RecordEnumeration` will update itself automatically. However, if the boolean value is set to `false`, the `RecordEnumeration` will not update itself until the `rebuild( )` method is called. Which method to use is entirely your choice. However, keep in mind that each call to `enumerateRecords( )` can take quite a bit of time to complete. If you have a large data store, it may be better to implement another strategy.

The following methods of `RecordEnumeration` will also set and retrieve the boolean property to keep the enumeration updated with any changes to the record store:

```
public void keepUpdated(boolean keepUpdated);  
public boolean isKeptUpdated( );
```

To find out how many records have been filtered into the enumeration itself, use the following method. (Note the number of filtered records may not be the same as the total amount of records in the record store.)

```
public int numRecords( );
```

If you wish to find out if the enumeration has an element before or after the current position, you can use the following methods. Note that the enumeration will loop around in the event that it travels before the first element or after the final element.

```
public boolean hasPreviousElement( );  
public boolean hasNextElement( );
```

If you wish to find the record ID that the previous methods refer to, use these methods:

```
public int nextRecordId( ) throws InvalidRecordIDException;  
public int previousRecordID( ) throws InvalidRecordIDException;
```

You can obtain the next and previous records themselves using the following methods:

```
public byte[] nextRecord( ) throws InvalidRecordIDException,  
    RecordStoreNotOpenException, RecordStoreException;  
public byte[] previousRecord( ) throws InvalidRecordIDException,  
    RecordStoreNotOpenException, RecordStoreException;
```

This method returns the current position of the enumeration to the state it was at when the enumeration was first created:

```
public void reset( );
```

And finally, if you want to release all the data that has been stored in the enumeration in order to gain back the memory used, utilize the following method:

```
public void destroy( );
```

8.2.6 Listening to Record Stores

If you wish to monitor any changes that take place in a record store, you can create an object that implements the `javax.microedition.rms.RecordListener` interface. This interface contains only three methods:

```
public interface RecordListener {
    public void recordAdded(RecordStore recordStore, int recordID);
    public void recordRemoved(RecordStore recordStore, int recordID);
    public void recordModified(RecordStore recordStore, int recordID);
}
```

As you probably guessed, the first method will be called if a record is added, the second will be called if a record is removed, and the third method will be called if a record is modified. All three methods have the same two parameters. The first is a reference to the `RecordStore` object, and the second parameter is an integer that refers to the actual record store ID that was changed.

Once you've created an object that implements the `RecordListener` interface, you can add or remove it from the list of listeners of a record store by using the following two methods of the `RecordStore` class:

```
public void addRecordListener(RecordListener listener);
public void removeRecordListener(RecordListener listener);
```

8.2.7 A Stock Database

Now let's see how we would use the RMS package to build a stock database. This example demonstrates how to work with the RMS to build a real MIDlet application. The application builds on the network programming experience we gained in [Chapter 7](#) and is similar to the `StockMIDlet` demo that comes with the MIDP. The MIDlet for this example does the following:

- Creates a record store (database).
- Adds new records (stocks) to the record store.
- Views the stocks in the database.

To add a stock to the database, the user enters the stock symbol (such as SUNW, IBM, IT, MS, GM, or Ford). The MIDlet retrieves the corresponding stock quote from Yahoo! Finance (<http://quote.yahoo.com>), constructs a record, and adds it to the database. To view the stocks in the database, the MIDlet iterates through the records

in the record store and prints them on the display in a nice format. The implementation of this MIDlet consists of the following three classes: `Stock.java`, `StockDB.java`, and `QuotesMIDlet.java`.

8.2.7.1 The `Stock.java` Class

This class parses a string obtained from Yahoo! Finance or from the record store into fields that represent, for example, the name of the stock or its price. The string returned from Yahoo! Finance has the following format:

```
NAME TIME PRICE CHANGELOWHIGHOPENPREV
    "SUNW", "2:1PM - <b>79.75</b>", +3.6875, "64.1875 -
129.3125", 78, 76.0625
```

In the MIDlet shown in [Example 8-1](#), the fields retrieved are the name of the stock, the time, and the price.

Example 8-1. Parses a string obtained from Yahoo! Finance or from a database

```
public class Stock {

    private static String name, time, price;

    // Given a quote from the server,
    // retrieve the name,
    // price, and date of the stock

    public static void parse(String data) {
        int index = data.indexOf('');
        name = data.substring(++index, (index = data.
            indexOf(' ', index)));
        index +=3;
        time = data.substring(index, (index = data.
            indexOf('-', index))-1);
        index +=5;
        price = data.substring(index, (index = data.
            indexOf('<', index)));
    }

    // Get the name of the stock from
    // the record store
    public static String getName(String record) {
        parse(record);
        return(name);
    }

    // Get the price of the stock from
    // the record store
    public static String getPrice(String record) {
        parse(record);
        return(price);
    }
}
```

8.2.7.2 The `StockDB.java` Class

This class provides methods that perform the following operations:

- Opens a new record store.
- Adds a new record to the record store.
- Closes the record store.
- Enumerates through the records.

Once you understand how to open a record store, add a new record, and close the record store, the code in [Example 8-2](#) is easy to follow.

Example 8-2. Provide methods for record store operations

```
import javax.microedition.rms.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

public class StockDB {
    RecordStore recordStore = null;
    public StockDB( ) {}

    // Open a record store with the given name
    public StockDB(String fileName) {
        try {
            recordStore = RecordStore.openRecordStore(fileName, true);
        } catch (RecordStoreException rse) {
            rse.printStackTrace( );
        }
    }

    // Close the record store
    public void close( ) throws RecordStoreNotOpenException,
        RecordStoreException {
        if (recordStore.getNumRecords( ) == 0) {
            String fileName = recordStore.getName( );
            recordStore.closeRecordStore( );
            recordStore.deleteRecordStore(fileName);
        } else {
            recordStore.closeRecordStore( );
        }
    }

    // Add a new record (stock)
    // to the record store

    public synchronized void addNewStock(String record) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream( );
        DataOutputStream outputStream = new DataOutputStream(baos);
        try {
            outputStream.writeUTF(record);
        } catch (IOException ioe) {
            System.out.println(ioe);
            ioe.printStackTrace( );
        }
        byte[] b = baos.toByteArray( );
        try {
            recordStore.addRecord(b, 0, b.length);
        } catch (RecordStoreException rse) {
            System.out.println(rse);
            rse.printStackTrace( );
        }
    }
}
```

```

    }

    // Enumerate through the records.
    public synchronized RecordEnumeration enumerate( ) throws
        RecordStoreNotOpenException {
        return recordStore.enumerateRecords(null, null, false);
    }
}

```

8.2.7.3 The QuotesMIDlet.java Class

Finally, the `QuotesMIDlet` class is the actual MIDlet that performs the following tasks:

- Create commands (List Stocks, Add a New Stock, Back, Save, Exit)
- Handle command events
- Connect to Yahoo! Finance and retrieve quotes
- Invoke methods from `Stock` and `StockDB` to parse quotes and add new stocks to the record store

The source for this file is listed in [Example 8-3](#).

Example 8-3. A MIDlet for the stock database

```

import javax.microedition.rms.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.io.*;
import java.io.*;
import java.util.Vector;

public class QuotesMIDlet extends MIDlet implements CommandListener {
    Display display = null;
    List menu = null; // main menu
    List choose = null;
    TextBox input = null;
    Ticker ticker = new Ticker("Database Application");
    String quoteServer =
        "http://quote.yahoo.com/d/quotes.csv?s=";
    String quoteFormat = "&f=slclwop"; // The only format supported

    static final Command backCommand = new Command("Back",
        Command.BACK, 0);
    static final Command mainMenuCommand = new Command("Main",
        Command.SCREEN, 1);
    static final Command saveCommand = new Command("Save",
        Command.OK, 2);
    static final Command exitCommand = new Command("Exit",
        Command.STOP, 3);
    String currentMenu = null;

    // Stock data
    String name, date, price;

    // record store
    StockDB db = null;

    public QuotesMIDlet( ) { // constructor

```

```

    }

    // start the MIDlet
    public void startApp( ) throws MIDletStateChangeException {
        display = Display.getDisplay(this);
        // open a db stock file
        try {
            db = new StockDB("mystocks");
        } catch(Exception e) {}
        menu = new List("Stocks Database", Choice.IMPLICIT);
        menu.append("List Stocks", null);
        menu.append("Add A New Stock", null);
        menu.addCommand(exitCommand);
        menu.setCommandListener(this);
        menu.setTicker(ticker);

        mainMenu( );
    }

    public void pauseApp( ) {
        display = null;
        choose = null;
        menu = null;
        ticker = null;

        try {
            db.close( );
            db = null;
        } catch(Exception e) {}
    }

    public void destroyApp(boolean unconditional) {
        try {
            db.close( );
        } catch(Exception e) {}
        notifyDestroyed( );
    }

    void mainMenu( ) {
        display.setCurrent(menu);
        currentMenu = "Main";
    }

    // Construct a running ticker
    // with stock names and prices
    public String tickerString( ) {
        StringBuffer ticks = null;
        try {
            RecordEnumeration enum = db.enumerate( );
            ticks = new StringBuffer( );
            while(enum.hasNextElement( )) {
                String stock1 = new String(enum.nextRecord( ));
                ticks.append(Stock.getName(stock1));
                ticks.append(" @ ");
                ticks.append(Stock.getPrice(stock1));
                ticks.append(" ");
            }
        } catch(Exception ex) {}
        return (ticks.toString( ));
    }
}

```

```

// Add a new stock to the record store
// by calling StockDB.addNewStock( )
public void addStock( ) {
    input = new TextBox("Enter a Stock Name:", "", 5,
        TextField.ANY);
    input.setTicker(ticker);
    input.addCommand(saveCommand);
    input.addCommand(backCommand);
    input.setCommandListener(this);
    input.setString("");
    display.setCurrent(input);
    currentMenu = "Add";
}

// Connect to quote.yahoo.com and
// retrieve the data for a given
// stock symbol.
public String getQuote(String input) throws IOException,
    NumberFormatException {
    String url = quoteServer + input + quoteFormat;
    StreamConnection c = (StreamConnection)Connector.open(
        url, Connector.READ_WRITE);
    InputStream is = c.openInputStream( );
    StringBuffer sb = new StringBuffer( );
    int ch;
    while((ch = is.read( )) != -1) {
        sb.append((char)ch);
    }
    return(sb.toString( ));
}

// List the stocks in the record store
public void listStocks( ) {
    choose = new List("Choose Stocks", Choice.MULTIPLE);
    choose.setTicker(new Ticker(tickerString( )));
    choose.addCommand(backCommand);
    choose.setCommandListener(this);
    try {
        RecordEnumeration re = db.enumerate( );
        while(re.hasNextElement( )) {
            String theStock = new String(re.nextRecord( ));
            choose.append(Stock.getName(theStock)+" @ " +
                Stock.getPrice(theStock),null);
        }
    } catch(Exception ex) {}
    display.setCurrent(choose);
    currentMenu = "List";
}

// Handle command events
public void commandAction(Command c, Displayable d) {
    String label = c.getLabel( );
    if (label.equals("Exit")) {
        destroyApp(true);
    } else if (label.equals("Save")) {
        if(currentMenu.equals("Add")) {
            // add it to database
            try {
                String userInput = input.getString( );
                String pr = getQuote(userInput);
                db.addNewStock(pr);
            }
        }
    }
}

```

```

        ticker.setString(tickerString( ));
    } catch(IOException e) {
    } catch(NumberFormatException se) {
    }
    mainMenu( );
}
} else if (label.equals("Back")) {
    if(currentMenu.equals("List")) {
        // go back to menu
        mainMenu( );
    } else if(currentMenu.equals("Add")) {
        // go back to menu
        mainMenu( );
    }
} else {
    List down = (List)display.getCurrent( );
    switch(down.getSelectedIndex( )) {
        case 0: listStocks( );break;
        case 1: addStock( );break;
    }
}
}
}
}

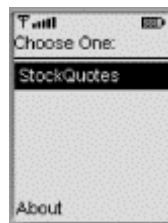
```

8.2.8 Testing QuotesMIDlet

To test QuotesMIDlet, use the J2ME Wireless Toolkit as we have throughout the book:

1. Create a new project and compile the code.
2. Run the MIDlet in the emulator. You should see QuotesMIDlet running in the emulator, as shown in [Figure 8-1](#).

Figure 8-1. QuotesMIDlet



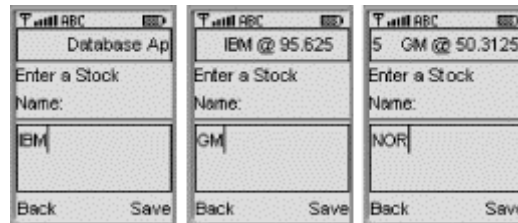
3. Activate QuotesMIDlet. You should see a menu with the following two options: List Stocks, and Add a New Stock, as shown in [Figure 8-2](#).

Figure 8-2. QuotesMIDlet stock database



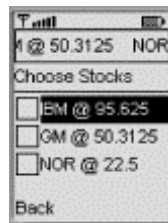
4. Choose the Add a New Stock option and add a few stocks. [Figure 8-3](#) shows that the stocks IBM, GM, and NOR were added in this example.

Figure 8-3. Adding new stocks



5. Go back and choose the View Stocks option. [Figure 8-4](#) shows that this option reads the record store and retrieves all the records (stocks) that have been added.

Figure 8-4. Viewing the record store



Have fun keeping track of your stocks! In the next chapter, we will discuss how to install MIDlets such as this on Palm computing platforms.

Chapter 9. The MIDP for Palm OS

An early access release of the MIDP for Palm OS was released just before JavaOne 2001, in early June, and the First Customer Shipping (FCS) of the MIDP for Palm OS 1.0 was released in mid-October, 2001. The MIDP for Palm OS is a J2ME application runtime environment based on the CLDC 1.0 and MIDP 1.0 specifications. It is targeted at handheld devices (such as Palm Pilot, Handspring Visor, and so on) running Palm OS version 3.5 or higher.

This chapter explains how to install the MIDP for Palm OS on your handheld device, and then how to convert existing MIDlets, developed in earlier chapters, into Palm Resource Code (PRC) files (executable Palm OS applications). The J2ME Wireless Toolkit 1.0.3 supports the MIDP for Palm OS. Hence, it is possible to test MIDlets using a Palm OS device, as shown in [Chapter 4](#).

9.1 Installing the MIDP for Palm OS on the Windows Platform

To install the MIDP for Palm OS, you need to perform the following steps:

1. Download the MIDP for Palm OS. This package (*midp4palm-1_0.zip*), which is less than one megabyte, is the early access release of the MIDP for Palm OS

implementation. You can download it from the Java web site at <http://java.sun.com/products/midp4palm>.

2. Unzip the package. Do this in the root directory *C:*. This gives you a new folder called *midp4palm1.0* that contains tools and sample applications. Check to make sure you have a file called *MIDP.prc* in the *PRCfiles* directory, which is the application runtime environment that supports the MIDP for Palm OS.

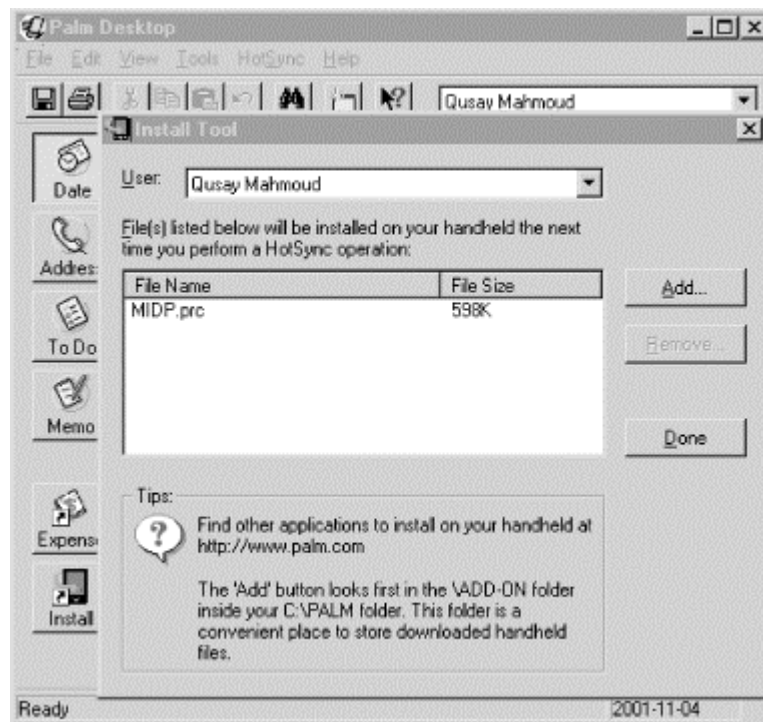
At this point, you are ready to install the MIDP for Palm OS on your Palm device.

9.1.1 Installing the MIDP for Palm OS on the Device

Use the HotSync application that came with the PalmPilot to install the *MIDP.prc* on your Palm OS device.

1. Place your Palm device in the cradle.
2. Using your Palm Desktop software on the PC (or a similar program), click the Install icon and browse to *C:\midp4palm1.0\PRCfiles* to select *MIDP.prc*, as shown in [Figure 9-1](#). Press the HotSync button on the cradle to install the file.

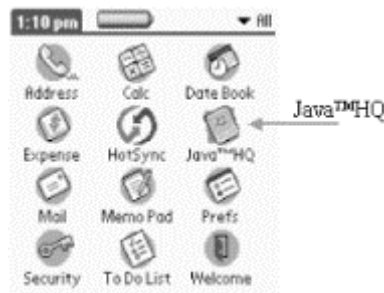
Figure 9-1. Using Palm Desktop to install MIDP.prc



3. Go to the application directory on your Palm and check to see if the Java HQ is there. The Java HQ is the application runtime environment that supports the MIDP for Palm OS. Note that the Java Manager environment takes up roughly 600K of your Palm OS device's storage.

You should see the Java HQ icon, as shown in [Figure 9-2](#). If you do not see it, click on another application (e.g., Calculator) and then go back to the application directory to reset the display.

Figure 9-2. Java HQ special icon on Palm



If you tap the Java HQ icon, you'll see the About screen, as shown in [Figure 9-3](#). This contains the copyright notice and gives you the option to change the Java HQ preferences that apply to every Java application running on the device. More information on the Java HQ preference settings is presented later in this chapter.

Figure 9-3. Java HQ About screen



Now you are ready to install other Java applications on your Palm.

9.1.2 Running Sample Applications

There are several applications for the Palm that are written in Java and that come with the software you just installed. All the files with the *.prc* extension in the directory *C:\midp4palm1.0\PRCfiles* are MIDP-based Java applications for the Palm. You can use the HotSync to install them.

As an example, use the HotSync and install the files *Games.prc* and *Demos.prc*. Once you have installed these two files, you will see two new icons, as shown in [Figure 9-4](#).

Figure 9-4. Installing sample MIDP-based Java applications



A MIDP-based Java application for the Palm runs just like any other Palm application. Simply tap the corresponding icon. Note that the Java HQ must already be installed. When you launch a Java application on the Palm, the Java HQ (which is

equivalent to the application manager) runs automatically. Also, it is important to note that the first time you launch an application, you are asked to read and accept the MIDP license; the application starts automatically once you tap Accept, as shown in [Figure 9-5](#).

Figure 9-5. Running Java applications on the Palm for the first time



If you open an application suite (such as Demos), you are prompted to choose which application you want to run, as shown in [Figure 9-6](#). Select an application to run by tapping on its arrow.

Figure 9-6. Selecting an application to run from a suite



9.2 Developing New Applications

You can easily develop new Java applications for the Palm if you are familiar with the MIDlet programming model. If you have run the MIDlets in earlier chapters, or have developed new ones, you can easily turn them into Palm applications by using a converter tool, which we will describe later in this chapter. The development life cycle of a Palm application can be summarized in the following three steps:

1. Develop a MIDlet or a MIDlet suite.
2. Convert the JAR/JAD file pair into a PRC file (executable Palm application).
3. Install the PRC file on the Palm and test the application.

9.2.1 Develop a MIDlet

As always, you can use either the Sun Microsystems Wireless Toolkit or your favorite development environment to develop a MIDlet. [Example 9-1](#) lists the MIDlet that was developed in [Chapter 6](#), which shows how to create various GUI components. The MIDlet for this example allows you to test lists, forms, choices, gauges, text fields, and text boxes.

Example 9-1. GuiTests.java

```

import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class GuiTests extends MIDlet implements CommandListener {

    // display manager
    Display display = null;

    // a menu with items
    List menu = null; // main menu

    // list of choices
    List choose = null;

    // textbox
    TextBox input = null;

    // ticker
    Ticker ticker = new Ticker("Test GUI Components");

    // alerts
    final Alert soundAlert = new Alert("sound Alert");

    // date
    DateField date = new DateField("Today's date: ",
        DateField.DATE);

    // form
    Form form = new Form("Form for Stuff");

    // gauge
    Gauge gauge = new Gauge("Progress Bar", false, 20, 9);

    // text field
    TextField textfield = new TextField("TextField Label", "abc",
        50, 0);

    // command
    static final Command backCommand = new Command("Back",
        Command.BACK, 0);
    static final Command mainMenuCommand = new Command("Main",
        Command.SCREEN, 1);
    static final Command exitCommand = new Command("Exit",
        Command.STOP, 2);
    String currentMenu = null;

    // constructor.
    public GuiTests( ) {
    }

    /**
     * Start the MIDlet by creating a list of items and associating
     * the exit command with it.
     */

    public void startApp( ) throws MIDletStateChangeException {
        display = Display.getDisplay(this);
        // open a db stock file

        menu = new List("Test Components", Choice.IMPLICIT);
        menu.append("Test TextBox", null);
    }

```

```

        menu.append("Test List", null);
        menu.append("Test Alert", null);
        menu.append("Test Date", null);
        menu.append("Test Form", null);
        menu.addCommand(exitCommand);
        menu.setCommandListener(this);
        menu.setTicker(ticker);

        mainMenu( );
    }

    public void pauseApp( ) {
        display = null;
        choose = null;
        menu = null;
        ticker = null;
        form = null;
        input = null;
        gauge = null;
        textfield = null;
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed( );
    }

    // main menu
    void mainMenu( ) {
        display.setCurrent(menu);
        currentMenu = "Main";
    }

    /**
     * Test the TextBox component.
     */
    public void testTextBox( ) {
        input = new TextBox("Enter Some Text:", "", 10,
            TextField.ANY);
        input.setTicker(new Ticker("Testing TextBox"));
        input.addCommand(backCommand);
        input.setCommandListener(this);
        input.setString("");
        display.setCurrent(input);
        currentMenu = "input";
    }

    /**
     * Test the List component.
     */
    public void testList( ) {
        choose = new List("Choose Items", Choice.MULTIPLE);
        choose.setTicker(new Ticker("Testing List"));
        choose.addCommand(backCommand);
        choose.setCommandListener(this);
        choose.append("Item 1", null);
        choose.append("Item 2", null);
        choose.append("Item 3", null);
        display.setCurrent(choose);
        currentMenu = "list";
    }
}

```

```

/**
 * Test the Alert component.
 */

public void testAlert( ) {
    soundAlert.setType(AlertType.ERROR);
    //soundAlert.setTimeout(20);
    soundAlert.setString("** ERROR **");
    display.setCurrent(soundAlert);
}

/**
 * Test the DateField component.
 */
public void testDate( ) {
    java.util.Date now = new java.util.Date( );
    date.setDate(now);
    Form f = new Form("Today's date");
    f.append(date);
    f.addCommand(backCommand);
    f.setCommandListener(this);
    display.setCurrent(f);
    currentMenu = "date";
}

/**
 * Test the Form component.
 */
public void testForm( ) {
    form.append(gauge);
    form.append(textfield);
    form.addCommand(backCommand);
    form.setCommandListener(this);
    display.setCurrent(form);
    currentMenu = "form";
}

/**
 * Handle events.
 */
public void commandAction(Command c, Displayable d) {
    String label = c.getLabel( );
    if (label.equals("Exit")) {
        destroyApp(true);
    } else if (label.equals("Back")) {
        if (currentMenu.equals("list") ||
            currentMenu.equals("input") ||
            currentMenu.equals("date") ||
            currentMenu.equals("form")) {
            // go back to menu
            mainMenu( );
        }
    } else {
        List down = (List)display.getCurrent( );
        switch(down.getSelectedIndex( )) {
            case 0: testTextBox( );break;
            case 1: testList( );break;
            case 2: testAlert( );break;
            case 3: testDate( );break;
            case 4: testForm( );break;
        }
    }
}

```

```

    }

}

}
}

```

9.2.2 Convert a MIDlet into a PRC file

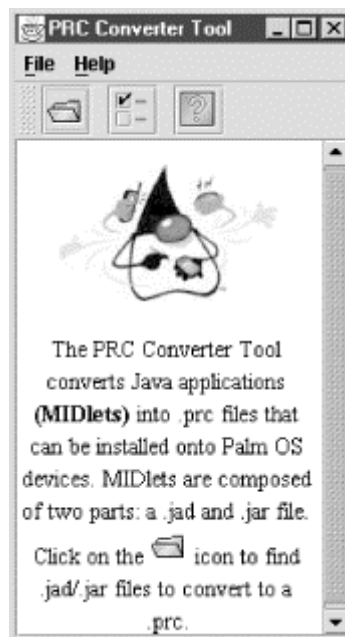
This section will explain how to convert a MIDlet file into a PRC file. First, build the MIDlet in [Example 9-1](#) and make sure there are no compilation errors. Most development tools will create the JAR and JAD files for you automatically. These are the two files needed to convert a MIDlet or a MIDlet suite into a PRC file. If you are using the Wireless Toolkit, the JAR and JAD files can be found in the *bin* directory of your project.

The MIDP for Palm OS comes with a converter tool to convert a MIDlet JAR/JAD into an executable Palm application. To run the PRC converter tool, you can use the batch file distributed with the release. However, if you have set the *JAVA_HOME* environment variable on your desktop, edit the *CONVERTER.BAT* file and change all the *JAVA_PATH* references to *JAVA_HOME*. Then run the converted batch file. Alternatively, you can run the tool using the command:

```
C:\midp4palm1.0\converter> java -jar Converter.jar
```

The *converter.jar* archive contains the implementations for the PRC converter tool. If the above command runs successfully, you should see a window similar to [Figure 9-7](#).

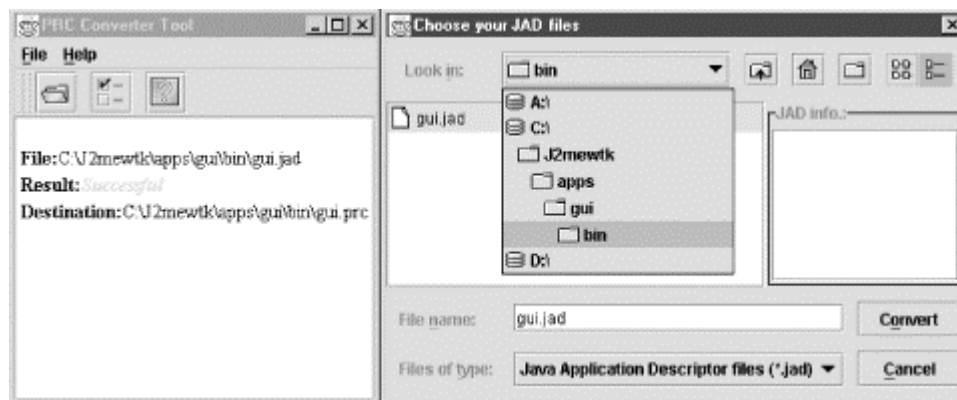
Figure 9-7. PRC converter tool



Now, select Convert from the File menu, and navigate to the directory where the JAD and JAR files are located (as with deployment on the Motorola i85s and the i50x, they must be in the same directory). Select the JAD file to be converted, then click on the

Convert button to convert the file into a PRC file. If everything is okay, you will see a success message, as shown in [Figure 9-8](#).

Figure 9-8. Converting JAD/JAR to PRC

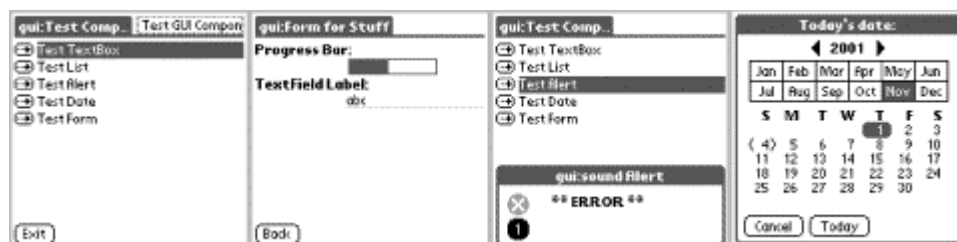


By default, the converted PRC file will be saved in the same directory as the JAD/JAR file pair. If you like, you can save all converted PRC files under another directory by choosing Preferences from the Converter's File menu. Then you can select a folder of your choice for output.

9.2.3 Install and Test

Once the JAD/JAR file pair have been converted to a PRC file, you can install the PRC file on your Palm OS device using HotSync. Once installed, you can run it and select components to test, as shown in [Figure 9-9](#). Here, we have tested a form with a progress bar, a text field, an alert, and a date.

Figure 9-9. Testing a MIDP application for the Palm OS



9.3 PRC Command-Line Conversion

The PRC GUI-based converter tool is easy to use. However, this comes at the expense of functionality. For example, what if you wish to associate a new icon with your application rather than have the default icon? You can use the command-line converter to complete this and other tasks.

The MIDP for Palm OS distribution comes with a command-line tool for converting JAR files to PRC files. The tool is the *MakeMIDPApp*, which is part of the *converter.jar* archive. To run this converter, use the command:

```
C:\mid4palm1.0\converter> java -cp converter.jar
com.sun.midp.palm.database.MakeMIDPApp
[options] JARfile
```

The options for this command are shown in [Table 9-1](#):

Table 9-1. Command-line PRC converter tool options	
Option	Description
<code>-v</code>	Verbose output
<code>-v -v</code>	More information
<code>-verbose</code>	Same as <code>-v</code>
<code>-icon <file></code>	File containing icon (in bmp, pbm, or bin Palm resource format) for the list view of the application
<code>-smallicon <file></code>	File containing a small icon for the Palm OS device's icon view
<code>-name <name></code>	Short name for the application, for the Palm OS device's icon view
<code>-longname <name></code>	Long name for the application, for Palm OS device list view
<code>-creator <criid></code>	Creator ID for the application
<code>-type <type></code>	Type file for the application (default is appl)
<code>-outfile <file></code>	Name of the PRC file to create
<code>-o <file></code>	Same as <code>-outfile</code>
<code>-version <string></code>	Change version
<code>-help</code>	Print help information
<code>-jad <JADfile></code>	A JAD file is specified (MIDlet suite packaging)

With the command-line tool, you can produce PRC files from a single MIDlet or from a MIDlet suite. For example, the following command can be used to convert a JAR file (containing one MIDlet or a MIDlet suite) to a PRC file:

```
C:\mid4palm1.0\converter> java -cp Converter.jar
com.sun.midp.database.MakeMIDPApp -type
Data gui.jar
```

This command will produce a PRC file called *gui.prc* from the JAR file *gui.jar*. Note that the type of application being converted can be either `appl` or `Data` (case-sensitive). If you don't provide the `-type` option, then `MakeMIDPApp` uses the default type, which is `appl`. It is important to note, however, that if you don't provide a creator ID with the `-creator` option, you must set the type to `Data`. The creator ID specifies the unique, four-character identifier for a Palm application. Every Palm application must have a creator ID, and if you do not provide one, then `MakeMIDPApp` will automatically generate a creator ID for your application. To find out the creator ID, use the `-v -v` option.

Any application converted using the GUI-based converter tool or the command-line tool is, by default, not beamable from the Palm launcher screen, as shown in [Figure 9-10](#). If you use the command-line tool, however, and provide a creator ID, then the application will be beamable.

Figure 9-10. An application cannot be beamed (by default)



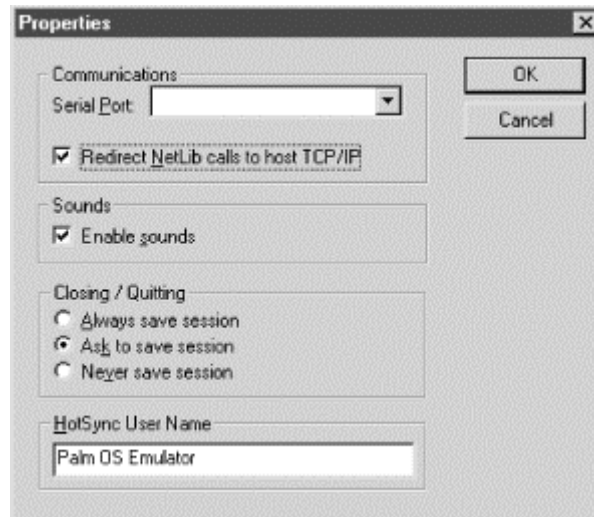
9.4 Advanced Java Applications

You have seen how to develop a simple Java application for the Palm that creates various GUI components. What about advanced applications that use networking and databases? Well, the MIDP for Palm OS supports all the MIDP features, including the Generic Connection Framework and the RMS. So, now let's look at a couple of sample applications developed in [Chapter 7](#) and [Chapter 8](#).

First, however, there are two things that need to be set if you want to test network-based applications from the Palm OS Emulator (POSE).

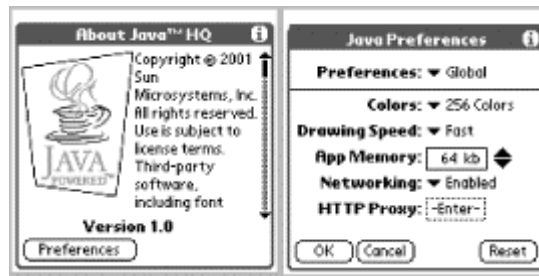
1. Redirect Netlib calls to host TCP/IP. To do this, right-click on the POSE window, select Setting → Properties, and check the Redirect NetLib calls to host TCP/IP, as shown in [Figure 9-11](#).

Figure 9-11. POSE property settings



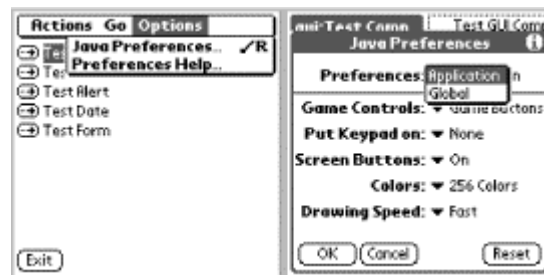
2. Enable Networking. To do this, tap the Java HQ icon, then tap Preferences and select Networking Enabled, as shown in [Figure 9-12](#).

Figure 9-12. Java HQ networking preferences



The Java HQ allows you to set special preferences. For example, it allows you to set how much memory is used to run Java applications, how many colors are used, the drawing speed, how your device will connect to the Internet, and how the controls should be displayed on the screen. You can easily set all of these options using the Java HQ. However, if you are running an application and would like to set some preferences, select the Preferences item from the Options menu. You can choose whether you want to set Application preferences, Global preferences, or Java HQ preferences, as shown in [Figure 9-13](#). Application preferences affect only the Java application you are running; Global preferences affect every Java application running in your device.

Figure 9-13. Preference settings for the Java HQ and its applications



9.4.1 Fetching a Page Using HttpURLConnection

In [Chapter 7](#), a MIDlet was developed to retrieve the contents of a file from a remote server using the `HttpURLConnection` interface. The MIDlet that implements this functionality is *SecondExample.java*. Create a new project in the Wireless Toolkit and use *SecondExample.java* as its source file. Build it, locate the JAR and JAD files, and use the PRC converter tool to convert them into a PRC file. Install the PRC file on POSE and then run the application. If all goes well, you should see something similar to [Figure 9-14](#).

Figure 9-14. Retrieving a file from a remote server



9.4.2 Retrieving Stock Quotes and Working with Databases

In [Chapter 8](#), a MIDlet was developed that allows you to create a database, add stocks (which are retrieved from Yahoo! Finance) to the database, and view the database. Download the files *Stock.java*, *StockDB.java*, and *QuotesMIDlet.java*. Build the application and create the JAR and JAD file pair. Then use the PRC converter tool to produce a PRC file. Install the PRC file on POSE and then run it. Add a few stocks and then view the stocks from the database. If everything goes well, you should see something similar to [Figure 9-15](#).

Figure 9-15. Stock quotes



9.5 A Final Thought

The Palm OS implementation of MIDP provides a runtime environment and tools that allow you to convert your MIDlets into Palm applications without writing a single line of code. I do not know of any easier way to develop fully functional Java applications for the Palm. These applications can be used just like any other Palm application; they run like any Palm program and can be removed from your Palm OS device the same way that you would remove any other application. However, it remains to be seen whether the MIDP for Palm OS is here to stay, or if it will be superseded by the PDA Profile for J2ME, which is based on the CLDC and will provide user interface and data storage APIs for handheld devices. As of this writing, the PDA profile is still in the works and no reference implementation is available.