BlackBerry Java Development Environment

Version 4.0

BlackBerry Application Developer Guide

Volume 2: Advanced Topics

BlackBerry Java Development Environment Version 4.0 BlackBerry Application Developer Guide Volume 2: Advanced Topics

Last modified: 26 November 2004

Part number: SWD_X_JDE(EN)-002.000

At the time of publication, this documentation complies with BlackBerry JDE Version 4.0.

Research In Motion Limited
295 Phillip Street
Waterloo, ON N2L 3W8
Canada

Research In Motion UK Limited
Centrum House, 36 Station Road
Egham, Surrey TW20 9LF
United Kingdom

Published in Canada

Contents

Using controlled APIs **5**
- BlackBerry controlled APIs 5
- Code signatures 6

Integrating email **11**
- BlackBerry mail API 11
- Working with messages 13
- Managing folders 18
- Managing attachments 19

Integrating PIM functions **23**
- PIM APIs 23
- Using the address book 25
- Using tasks 31
- Using the calendar 37

Adding handheld options **43**
- Options API 43
- Adding option items 43

BlackBerry Browser **47**
- Browser APIs 47
- Displaying web content 47
- Supporting additional MIME types 58
- Registering as a HTTP filter 64

Accessing the phone application **69**
- Using the phone API 69
- Listening for phone events 70
- Accessing and managing phone logs 70

Communicating with BlackBerry applications **75**
- Starting BlackBerry applications 75
- Adding menu items to BlackBerry applications 76

Storing persistent data **79**
- Storage options 79
- Managing persistent data 81
- Managing custom objects 85

Backing up and restoring persistent data **93**
- Synchronization API 93
- Adding support for backing up data 94

Accessing setup and configuration information 103
Service book API 103

Managing notifications 105
Notification API 105
Adding events 105
Responding to events 110
Customizing system notifications 112

Managing applications 119
Application manager 119
Managing code modules 121

Sharing runtime objects between applications 125
Sharing runtime objects 125

Glossary 128

Index 131

1

Using controlled APIs

- BlackBerry controlled APIs
- Code signatures

BlackBerry controlled APIs

The BlackBerry® APIs described in this guide have controlled access. Applications that use controlled APIs can be run in the simulator; however, you must obtain code signatures from Research In Motion (RIM) before you can load these applications onto BlackBerry Wireless Handhelds. See " Code signatures" on page 6 for more information.

Package	Description
`net.rim.blackberry.api.browser`	This package enables applications to invoke the BlackBerry Browser. See " Displaying web content" on page 47 for more information.
`net.rim.blackberry.api.invoke`	This package enables applications to invoke BlackBerry applications, such as tasks, messages, MemoPad and phone. See " Starting BlackBerry applications" on page 75 for more information.
`net.rim.blackberry.api.mail`	This package enables applications to interact with the BlackBerry messages application to send, receive, and open email messages. See " Working with messages" on page 13 for more information.
`net.rim.blackberry.api.mail.event`	This package defines messaging events and listener interfaces to manage mail events. See " Mail events" on page 13 for more information.
`net.rim.blackberry.api.menuitem`	This package enables you to add custom menu items to BlackBerry applications, such as the address book, calendar, and messages. See " Adding menu items to BlackBerry applications" on page 76 for more information.
`net.rim.blackberry.api.options`	This package enables you to add items to the handheld options. See " Adding option items" on page 43 for more information.
`net.rim.blackberry.api.pdap`	This package enables applications to interact with BlackBerry personal information management (PIM) applications, including address book, tasks, and calendar. Most of the same functionality is provided by the MIDP package `javax.microedition.pim`. See " PIM APIs" on page 23 for more information.
`net.rim.blackberry.api.phone`	This package provides access to advanced features of the phone application. See " Using the phone API" on page 69 for more information.
`net.rim.blackberry.api.phone.phonelogs`	This package provides access to the phone call history. See " Accessing and managing phone logs" on page 70 for more information.
`net.rim.device.api.browser.field`	This package enables applications to display a browser field within their user interface. See " Displaying web content in a browser field" on page 48 for more information.
`net.rim.device.api.browser.plugin`	This package enables you to add support for additional MIME types to the BlackBerry Browser. See " Supporting additional MIME types" on page 58 for more information.
`net.rim.device.api.crypto.*`	These packages provide data security capabilities, including data encryption and decryption, digital signatures, data authentication, and certificate management. See the *API Reference* for more information.
`net.rim.device.api.io.http`	This package enables applications to register with the BlackBerry Browser as provider for one or more URLs. See " Registering as a HTTP filter" on page 64 for more information.

Package	Description
net.rim.device.api.notification	This package provides methods to trigger event notifications and respond to system-wide and application-specific events. See " Notification API" on page 105 for more information.
net.rim.device.api.servicebook	This package enables applications to add, delete, and access service book entries. See " Accessing setup and configuration information" on page 103 for more information.
net.rim.device.api.synchronization	This package enables applications to perform backup and restore operations on custom data. See " Adding support for backing up data" on page 94 for more information.
net.rim.device.api.system	This package provides classes that enable functionality such as persistent data storage, interprocess communication (IPC), SMS, network communication using datagrams, and application management. See the following locations for more information: • "Application manager" on page 119 • "Using datagram connections" on page 96 • "Storing persistent data" on page 79

Code signatures

RIM tracks the use of some sensitive APIs in the BlackBerry JDE for security and export reasons. In the *API Reference*, a lock icon or the text "signed" indicates sensitive classes or methods. In the documentation for a class that contains signed methods, select or clear the **SHOW Signed** option at the top of the page to view or hide signed methods.

If you use signed classes or methods in your applications, the .cod files must be digitally signed by RIM before you can load them onto handhelds.

Note: To test and debug your code before receiving code signatures, use the simulator. Code must be signed only for deployment to handhelds.

Use the Signature Tool, which is installed with the BlackBerry JDE, to request the appropriate signatures for your .cod files.

Note: You never send your actual code to RIM. The Signature Tool sends an SHA-1 hash of your code file so that the signing authority system can generate the necessary signature.

Code signature verification

There are two types of code signature verification:

- **Linktime verification**: When you load a signed .cod file onto the handheld, the virtual machine (VM) links the .cod file with the API libraries and verifies that the .cod file includes the required signatures. If a signature is missing, the VM stops linking and does not load the application.
- **Runtime verification**: When the user uses the application on the handheld, if the application invokes a method that requires a signature, the VM verifies that the application contains the necessary signature. If the signature is not present, a `ControlledAccessException` is thrown and the requested operation is not performed.

See " Registering for code signing" on page 8 for more information on .csi files. See " Requesting code signatures" on page 9 for more information on .csl and .cso files.

Code signing request process

1. The Signature Tool opens an HTTP connection to the signing authority system and sends a request. The request includes a hash of your code in the .csl and .cso files. Your actual code is not sent to RIM.
2. The signing authority system verifies that the request is valid and applies a RIM private key to the hash of each .cod file to create the signatures.
3. The signing authority system returns the signatures to the Signature Tool and closes the HTTP connection.
4. The Signature Tool appends the signatures to each .cod file.

   When the files are signed, the Status column for the .cod file displays **Signed**.

   If any problems occur with the signature request, the Status column displays **Failed - See Details**.

When your .cod files are signed, you can load them onto the BlackBerry Wireless Handheld. See "Packaging and deploying applications" on page 32 of the *BlackBerry Application Developer Guide Volume 1 – Fundamentals* for more information.

Optional signatures

You can load applications onto handhelds without optional .cso signatures. These signatures are only required if their corresponding methods are invoked during runtime.

When the application calls a method that requires a signature, the VM verifies that the application has this authorization. If the VM does not find these optional signatures, the application stops.

Signing limitations

There are several situations in which the code signing process does not proceed.

Client parameters

The signing authority administrator can limit your access to signatures by specifying a limit using both time and frequency parameters. These parameters are defined in your .csi file. Be aware of these possible limitations when applying for signatures.

Parameter	Definition
# of Requests	This parameter sets the number of requests you can make using a particular .csi file. After you make the maximum number of requests, the .csi file is invalid and you can no longer make signature requests using this file. Contact your signing authority administrator to apply for another .csi file. Requests are limited for security reasons; however, the signing authority administrator can allow you to make an infinite number.
Expiry Date	This parameter sets the expiry date for your .csi file. After your .csi file expires, you can no longer make signature requests using this file. Contact your signing authority administrator and apply for another .csi file.

To request a change in these .csi parameters, contact your signing authority administrator.

Lost data

You cannot perform any code signing requests without your .csi file. Your registration key is stored within your .csi file — none of your signature requests can be sent to the signing authority system if the Signature Tool cannot find this key and sign your requests with it.

If your system stops responding, and you lose data or even entire file structures, you might discover that you have also lost the ability to perform signing requests. If you lose your .csi file, the Signature Tool cannot communicate with the signing authority system on your behalf.

If you lose your .csi file, contact your signing authority administrator and request a new one.

Registering for code signing

You require a separate set of code signing keys for each computer that requests keys. Once keys are installed on a computer, they cannot be reinstalled or moved to another computer.

Register for code signatures

You must have HTTP access to the Internet to register for code signing.

1. To activate your account, complete the registration form on the BlackBerry Developer Zone at http://www.blackberry.com/developers.

   In this form, you provide a 10-digit personal information number (PIN).
2. When you receive .csi files in an email message from RIM, save them to your computer.
3. Double-click a .csi file.

   If a dialog box appears that states that a private key cannot be found, perform the following actions before you continue:

   - Click **Yes** to create a new key pair file.
   - Type a password for your private key, and retype to confirm.
   - Click **OK**.
   - Move your mouse to generate data for a new private key.
4. In the **Registration PIN** field, type the PIN that RIM provided.
5. In the **Private Key Password** field, type a password of at least eight characters. This is your private key password, which protects your private key.

Note: Protect your private key password. If you lose this password, you must register with RIM again. If this password is stolen, contact RIM immediately to revoke your key to prevent others from requesting code signatures using your identity.

6. Click **Register**.
7. Click **Exit**.

Change your private key password

You must have HTTP access to the Internet to change your private key password.

1. In the BlackBerry JDE bin folder, double-click **SignatureTool.jar**.
2. Click **Change Password**.
3. In the **Old Password** field, type your current private key password.
4. Click **Verify**.
5. Type and confirm a new password.
6. Click **OK**.

Requesting code signatures

Request a code signature from the IDE

1. Build your projects.

   In the IDE, on the **Build** menu, click **Build All**. The IDE creates the following three files, located in the same folder as the project .jdp file, for each project:

File extension	Description
.cod file	the compiled project that is loaded on the handheld
.csl file	a list of required linktime signatures
.cso file	a list of signatures that might be required at runtime if the application invokes controlled methods

2. On the **Build** menu, click **Request signatures.**

Warning: If you have already registered for code signing with a previous version of the SDK, back up the following files, which are located in the BlackBerry JDE bin folder, before you install a new version of the BlackBerry JDE:

- Sigtool.db
- Sigtool.csk

If these files are lost, you must register again with RIM.

3. Click **Add**.
4. In the **Look In** drop-down list, click the folder in which the .cod file is located.
5. Click a .cod file.
6. Click **Open**.
7. Click **Request**.
8. Type your private key password.
9. Click **OK**.

Request signatures at a command prompt

1. At the command prompt, move to the folder containing the Signature Tool software.
2. Type the following command line:

   `java -jar SignatureTool.jar [-a] [-c] [-C] <file>`

   where:

   `[-a]` is used when you want the program to automatically request signatures.

   `[-c]` is used when you want the program to close after requesting signatures if no errors occur.

   `[-C]` is used when you want the program to close regardless of its success.

   `<file>` can be the name of only one .csi file or one or more .cod files.

   - **.csi:** The .csi file contains client registration information and a list of the signatures that the client is allowed to apply for. You can only pass in one .csi at a time.
   - **.cod:** The .cod file is the compiled application that can be loaded onto handhelds after all required signatures are in place. You can pass in as many .cod files as you want.

2

Integrating email

- BlackBerry mail API
- Working with messages
- Managing folders
- Managing attachments

BlackBerry mail API

The BlackBerry mail API, in the `net.rim.blackberry.api.mail` and `net.rim.blackberry.mail.event` packages, enables applications to send, receive, and access email messages using the messages application.

Notes: The BlackBerry mail API provides access to email messages in the handheld messages list, but not to other message types, such as SMS messages, PIN messages, or phone call logs. For access to phone call logs, use the Phone Log API (net.rim.blackberry.api.phonelogs). See " Accessing and managing phone logs" on page 70 for more information.

Check for a `ControlledAccessException` when your application first accesses the Mail API. This runtime exception is thrown if the system administrator restricts access to the Mail API using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

Mail API classes

Class name	Description
`Session`	This class, which represents an abstract model for email operations, provides access to email service, storage, and transport. Applications retrieve a new `Session` object to send or receive email messages. To retrieve a session with the default mail service on the handheld, invoke `Session.waitForDefaultSession()` and wait until the service is available. To retrieve a session with a different email service, create a `ServiceConfiguration` object for the email service and invoke `Session.getDefaultInstance(ServiceConfiguration)`.
`Store`	This class models the underlying message storage on the handheld. To retrieve a `Store` instance, invoke the `Session` instance.
`Transport`	This class represents the email transport protocol.

Messages

The `Message` class represents an email message. A `Message` object consists of a set of attributes, such as subject, sender, and recipients, and a message body (its contents). See " Multipart messages" on page 12 for more information.

The following classes and interfaces define supported message attributes:

Class or interface name	Description
`Address`	This class represents an email, ftp, http, or wap address that is used in the from, reply-to, and recipient attributes, and in the message body. The `Address` class contains fields to store the fully qualified address string, such as scott.tooke@rim.com, and the display name.
`Header`	This class defines supported header fields, such as `TO`, `FROM`, and `DATE`.
`Message.Flag`	This interface defines message flags, such as `MOVED`, `OPENED`, or `SAVED`.

Class or interface name	Description
Message.Icons	This interface defines the character representations of the various message status icons, such as a check mark for a sent message.
Message.RecipientType	This interface defines supported recipient types, such as TO, CC, or BCC.
Message.Status	This interface defines status options for sending and receiving messages, such as RX_RECEIVED, RX_ERROR, TX_SENT, and TX_READ.

Multipart messages

The mail API supports multipart messages. The Multipart abstract class provides a container for multiple BodyPart objects. Multipart provides methods to retrieve and set its subparts.

Each BodyPart consists of header fields (attributes) and contents (body). The mail API provides four implementations of BodyPart.

Class name	Description
ContactAttachmentPart	This class represents an address card attachment part, using the javax.microedition.pim.Contact interface. See " Using the address book" on page 25 for more information.
TextBodyPart	This class represents a body part with content that is text/plain type. You use this class to create a multipart message that includes a text/plain part.
UnsupportedAttachmentPart	This class represents an unsupported attachment part. You cannot instantiate this class. The content type is always application/octet-stream.
SupportedAttachmentPart	This class represents a supported attachment part, for which there is a registered attachment handler on the handheld.

Message storage

The Folder class represents a local mailbox folder. Several folder types are defined, including INBOX, OUTBOX, SENT, and OTHER. You can use these folder types to retrieve folders for retrieving or saving messages.

The Store class models the underlying message storage and provides methods for finding and retrieving folders. Folders exist in a hierarchy, as the following example demonstrates:

```
Mailbox - Ming Li
   Inbox
   Projects
      In Progress
      Complete
   Personal
```

A standard delimiter character separates each folder in the hierarchy, which you can retrieve using getSeparator(). You can list all the folders in a Store object, list the subfolders in a folder, or find a folder based on a search string.

The Folder class defines methods for retrieving messages or subfolders, saving messages, and deleting messages.

Note: Multiple Folder instances can refer to the same folder on the handheld. As a result, you should always invoke addFolderListener() and removeFolderListener() on the same Folder object. Use Folder.equals() to determine whether two Folder objects refer to the same folder.

Mail events

The BlackBerry mail event package (`net.rim.blackberry.api.mail.event`) defines the following messaging events, and listeners for each event:

Event	Description
`FolderEvent`	This event triggers when a message in a folder is added or removed.
`MessageEvent`	This event triggers when a message changes (body, header, or flags).
`StoreEvent`	This event triggers when a message is added to, or removed from, the message store in a batch operation (for example, when the handheld is synchronized with the desktop).

The `MailEvent` class is the base class for these mail event classes. The `MailEvent` class defines an abstract `dispatch()` method to invoke the appropriate listener method for each event.

The `EventListener` interface provides a common interface for the `FolderListener` and `MessageListener` interfaces.

The following table lists the objects to which each listener type can be added:

Listener	Applicable objects
`FolderListener`	`Folder` or `Store` object
`MessageListener`	`Message` object
`StoreListener`	`Store` object

Working with messages

Receive message notification

To receive message notification, implement the `FolderListener` and `StoreListener` interfaces.

```
public class MailTest implements FolderListener, StoreListener { ... }
```

Add a listener to the message store

To listen for message store events, such handheld synchronization, retrieve the `Store` object and add a `StoreListener` instance to it.

```
try {
   Store store = Session.waitForDefaultSession().getStore();
} catch (NoSuchServiceException e) {
   System.out.println(e.toString());
}
store.addStoreListener(this);
```

To define application behavior when messages are added to or removed from the message store in batch operations, implement `StoreListener.batchOperation()`. For example, your application could check if any messages to which it has references were removed.

```
void batchOperation(StoreEvent e) {
   //perform action when messages added or removed in batch operation
}
```

Add a listener to a folder

To listen for folder events, such as the addition of a message to a particular folder, retrieve the `Folder` object for which you want to receive notifications of new messages. Add the `FolderListener` instance to the folder.

```
Folder[] folders = store.list(Folder.INBOX);
Folder inbox = folders[0];
inbox.addFolderListener(this);
```

To perform actions when folder events occur, implement `FolderListener.messagesAdded()` and `FolderListener.messagesRemoved()`. For example, you could implement these methods to maintain the consistency of any references in your application to specific mail folders.

```
void messagesAdded(FolderEvent e) {
   //perform processing on added messages
}
void messagesRemoved(FolderEvent e) {
   //perform processing on removed messages
}
```

Receive more of a message

By default, the first section of a message (typically about 2 KB) is sent to the handheld. Invoke `hasMore()` on a body part to determine if more data is available on the server. Invoke `moreRequestSent()` to determine if a request for more data has already been sent. Invoke `more()` to request more of a message.

```
if (( bp.hasMore() ) && (! bp.moreRequestSent()) {
   Transport.more(bp, true);
}
```

The second parameter of `more()` is a Boolean value that specifies whether to retrieve only the next section of the body part (`false`) or all remaining sections of the body part (`true`).

Open a message

Retrieve the message store and the folder that contains the message.

```
Store store = Session.waitForDefaultSession.getStore();
Folder folder = Store.getFolder("SampleFolder");
```

Retrieve the message objects from the folder. Iterate through the array and retrieve information, such as the sender and subject, to display to the user.

```
Message[] msgs = folder.getMessages();
```

When a user selects a message from the list, invoke methods on the `Message` object to retrieve the appropriate fields and body contents to display to the user.

```
Message msg = msgs[0]; // retrieve the first message
Address[] recipients = msg.getRecipients(Message.RecipientType.TO)
Date sent = msg.getSentDate();
Address from = msg.getFrom();
String subject = msg.getSubject();
```

```
Object o = msg.getContent();
//verify that the message is not multipart
if ( o instanceof String ) {
   String body = (String)o;
}
...
```

Tip: Invoke `getBodyText()` on a message to retrieve the plain text contents as a `String`. If the message does not contain plain text, the method returns null.

Send a message

To send messages, use a `Transport` object, which represents the email transport protocol.

Create a message

Create a `Message` object, and specify a folder in which to save a copy of the sent message.

```
Store store = Session.getDefaultInstance().getStore();
Folder[] folders = store.list(Folder.SENT);
Folder sentfolder = folders[0];
Message msg = new Message(sentfolder);
```

Specify recipients

Create an array of `Address` objects and add each address to the array. You should catch an `AddressException`, which is thrown if an address is invalid.

```
Address toList[] = new Address[1];
try {
   toList[0]= new Address("scott.tooke@rim.com", "Scott Tooke");
} catch(AddressException e) {
   System.out.println(e.toString());
}
```

Add recipients

Invoke `Message.addRecipients()`. As parameters to this method, provide the type of recipient (`TO`, `CC`, or `BCC`) and the array of addresses to add. If your message has multiple types of recipients, invoke `addRecipients()` once each.

```
msg.addRecipients(Message.RecipientType.TO, toList);
```

Specify the name and email address of a sender

Invoke `setFrom()`.

```
Address from = new Address("scott.mcpherson@blackberry.com", "Scott McPherson");
msg.setFrom(from);
```

Add a subject line

Invoke `setSubject()`.

```
msg.setSubject("Test Message");
```

Specify the message contents

Invoke `setContent()`. Typically, you retrieve content from text that a user types in a field.

```
try {
   msg.setContent("This is a test message.");
} catch(MessagingException e) {
   System.out.println(e.getMessage());
}
```

Send the message

Invoke `Transport.send()`.

```
try {
   Transport.send(msg);
} catch(MessagingException e) {
   System.out.println(e.getMessage());
}
```

Reply to a message

To create a message as a reply to an existing message, invoke `Message.reply()`. As a parameter to this method, specify `true` to reply to all message recipients or `false` to reply only to the sender.

```
Store store = Session.waitForDefaultSession().getStore();
Folder[] folders = store.list(INBOX);
Folder inbox = folders[0];
Message[] messages = folder.getMessages();
if( messages.length > 0 ) {
   Message msg = messages[0];
}
Message reply = msg.reply(true);
Transport.send(reply);
```

Forward a message

Invoke `forward()` on an existing `Message` object.

Note: The subject line of a forwarded message is set automatically to **FW:<*original_subject*>**.

```
Message fwdmsg = msg.forward();
```

Add recipients

Create an array of addresses and invoke `addRecipients()`.

```
Address toList[] = new Address[1];
toList[0]= new Address("katie.laird@rim.com", "Katie Laird");
fwdmsg.addRecipients(Message.RecipientType.TO, toList);
```

Specify message contents

Invoke `setContent()`.

Note: You cannot edit the text of the forwarded message. The `setContent()` method adds text before the forwarded message.

```
try {
   fwdmsg.setContent("This is a forwarded message.");
} catch(MessagingException e) {
   System.out.println(e.getMessage());
}
```

Send the message

Invoke `send()`.

```
try {
   Transport.send(fwdmsg);
} catch(MessagingException e) {
   System.out.println(e.getMessage());
}
```

Code example

Example: BasicMail.java

```
/**
 * BasicMail.java
 * Copyright (C) 2001-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.basicmail;

import net.rim.blackberry.api.mail.*;
import net.rim.blackberry.api.mail.event.*;
import net.rim.device.api.system.*;

public class BasicMail extends Application {
    private Store store;
    static void main (String args[]) {
        BasicMail app = new BasicMail();
        app.enterEventDispatcher();
    }
    BasicMail() {
        Store store = Session.getDefaultInstance().getStore();
        Folder[] folders = store.list(Folder.SENT);
        Folder sentfolder = folders[0];
        // Create message.
        Message msg = new Message(sentfolder);
        // Add TO Recipients.
        Address toList[] = new Address[1];
        try {
            toList[0]= new Address("scott.tooke@rim.com", "Scott Tooke");
        } catch(AddressException e) {
            System.out.println(e.toString());
        }
        try {
            msg.addRecipients(Message.RecipientType.TO, toList);
        } catch (MessagingException e) {
            System.out.println(e.toString());
```

```
        }
        // Add CC Recipients.
        Address ccList[] = new Address[1];
        try {
            ccList[0]= new Address("katie.laird@rim.com", "Katie Laird");
        } catch(AddressException e) {
            System.out.println(e.toString());
        }
        try {
            msg.addRecipients(Message.RecipientType.CC, ccList);
        } catch (MessagingException e) {
            System.out.println(e.toString());
        }
        // Add the subject.
        msg.setSubject("A Test Email");
        // Add the message body.
        try {
            msg.setContent("This is a test message.");
        } catch(MessagingException e) {
                // Handle messaging exceptions.
        }
        // Send the message.
        try {
            Transport.send(msg);
        } catch(MessagingException e) {
                System.out.println(e.getMessage());
        }
        System.out.println("Email sent successfully.");
        System.exit(0);

    }
}
```

Managing folders

To list, retrieve, and search for folders, retrieve a `Store` object by invoking `getStore()` on the default session.

```
Store store = Session.waitForDefaultSession().getStore();
```

List folders

To list the folders in a mailbox store, invoke `Store.list()`.

```
Folder[] folders = store.list();
```

Retrieve an array of folders by type

Invoke `list()`. Provide the folder type as a parameter to this method.

```
Folder[] folders = store.list(INBOX);
Folder inbox = folders[0];
```

Retrieve an array of folders by searching

To retrieve all the folders in the hierarchy that match a specified search string, invoke `findFolder()`.

```
Folder[] folders = store.findFolder("Inbox");
```

The `findFolder()` method returns an array of folders that match the specified string, or an empty array if a matching folder is not found.

Retrieve a folder by name

Invoke `getFolder()`. Provide as parameter the absolute path to the folder. A `FolderNotFoundException` exception is thrown if the folder is not found.

```
Folder folder = store.getFolder("Mailbox - Scott Tooke/Inbox/Projects");
```

Retrieve a folder by ID

Invoke `getID()` to retrieve the folder ID, and then invoke `getFolder()` with the ID as a parameter.

```
Folder[] folders = store.list();
long id = folders[0].getId();
Folder f2 = store.getFolder(id);
```

File a message

Invoke `appendMessage()` on a `Folder` object.

```
Message msg = new Message();
...
Folder folder = store.getFolder("Inbox");
folder.appendMessage(msg);
```

Managing attachments

The mail API enables you to open incoming email attachments and to create outgoing attachments on the handheld. A message attachment is represented as a separate `BodyPart` on a `Multipart` message.

Create a custom attachment handler

Implement the `AttachmentHandler` interface.

Register accepted MIME types

To register the MIME type of attachments that your attachment handler accepts, implement `supports()`. This method is invoked when the handheld receives an attachment.

```
public boolean supports(String contentType) {
```

```
    return (contentType.toLowerCase().indexOf("contenttype") != -1 ? true : false);
}
```

Define the associated menu item string

Implement `menuString()` to return the menu item string to display in the messages list when the user selects an attachment.

```
public String menuString() {
    return "Custom Attachment Viewer";
}
```

Define attachment processing

Implement `run()` to perform the appropriate processing on the attachment data and display it to the user.

```
public void run(Message m, SupportedAttachmentPart p) {
    //perform processing on data
    Screen view = new Screen();
    view.setTitle(new LabelField("Attachment Viewer"));
    view.add(new RichTextField(new String((byte[])p.getContent())));
}
```

Note: The `run()` method is invoked when the corresponding menu item is selected in the messages list.

Register an attachment handler

The `AttachmentHandlerManager` class controls how attachments are processed on the handheld. To enable the message application to invoke your custom attachment when the user opens an attachment of the associated type, register your attachment handler by invoking `addAttachmentHandler()`.

Note: The BlackBerry Enterprise Server Attachment Service has first priority in receiving attachments. Third-party attachment handlers cannot override default handheld behavior. See the *BlackBerry Enterprise Server Maintenance and Troubleshooting Guide* for more information.

```
AttachmentHandlerManager m = AttachmentHandlerManager.getInstance();
CustomAttachmentHandler ah = new CustomAttachmentHandler();
m.addAttachmentHandler(ah);
```

Retrieve attachments

The `SupportedAttachmentPart` class represents an attachment with a corresponding viewer on the handheld. An attachment that does not have a viewer on the handheld is represented as an `UnsupportedAttachmentPart`.

Retrieve attachment contents

Invoke `getContent()`.

```
String s = new String((byte[])p.getContent());
```

Retrieve attachment information

The `SupportedAttachmentPart` class provides several methods on a to retrieve attachment information. The following example invokes `getName()` and `getSize()` to retrieve the attachment name and size.

```
public void run(Message m, SupportedAttachmentPart p) {
   ...
   String name = p.getName();
   int size = p.getSize();
}
```

Send an attachment

To send an email message with an attachment, create a multipart message by creating a new `Multipart` object. To create each attachment component, create a `SupportedAttachmentPart` object, designating the `Multipart` as its parent. To add each `SupportedAttachmentPart` to the `Multipart`, invoke `addBodyPart()` on that object.

When you invoke `setContent()` on the `Message` object, pass in the `Multipart` object as its parameter.

```
byte[] buf = new byte[256]; // the attachment
MultiPart multipart = new MultiPart(); // default type of multipart/mixed
SupportedAttachmentPart attach = new SupportedAttachmentPart( multipart,
   "application/x-example", "filename", data);
multipart.addBodyPart(attach); // add the attachment to the multipart
msg.setContent(multipart);
Transport.send(msg); // send the message
```

3

Integrating PIM functions

- PIM APIs
- Using the address book
- Using tasks
- Using the calendar

PIM APIs

The Java personal information management (PIM) APIs (`javax.microedition.pim`) and the BlackBerry personal digital assistant profile (PDAP) APIs (`net.rim.blackberry.api.pdap`) enable you to access the calendar, tasks, and address book on the handheld.

Note: As of version 4.0, the `net.rim.blackberry.api.pim` package is deprecated. The classes from this package are now available in the `javax.microedition.pim` and `net.rim.blackberry.api.pdap` packages.

The `PIM` class is an abstract class that provides methods for accessing PIM databases on the handheld. Invoke `PIM.getInstance()` to retrieve a PIM object.

Note: Check for a `ControlledAccessException` when your application first accesses the PIM API. This runtime exception is thrown if the system administrator restricts access to the PIM API using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

PIM lists

The `PIMList` interface represents the common functionality of all contact, event, or task lists. A list contains zero or more items, represented by subclasses of `PIMItem`. Use PIM lists to organize related items and to retrieve some or all of the items in the list.

Note: On handhelds, each `ContactList`, `ToDoList`, or `EventList` instance refers to the native database on the handheld. Third-party applications cannot create custom lists.

PIM items

The `PIMItem` interface represents the common functionality of an item in a list. The `Contact`, `Event`, and `ToDo` interfaces extend `PIMItem`. A PIM item represents a collection of data for a single entry, such as a calendar appointment or a contact.

When you create a PIM item in a particular PIM list, the item remains associated with that list as long as it exists. You can also import or export data in PIM items using standard formats, such as iCal and vCard.

Fields

A PIM item stores data in fields.

Each `PIMItem` interface—`Contact`, `Event`, or `ToDo`—defines unique integer IDs for each field that it supports. For example, the `Contact` interface defines fields to store an email address (`EMAIL`), name (`FORMATTED_NAME`), and phone number (`TEL`).

Each field has a data type ID, such as INT, BINARY, DATE, BOOLEAN, or STRING. To retrieve the data type of a field, invoke PIMList.getFieldDataType(int field). The data type determines which method you use to get or set field data. For example, if the data type for a field is STRING, invoke PIMItem.addString() to add a value, PIMItem.setString() to change an existing value, and PIMItem.getString() to retrieve a value.

Before you attempt to set or retrieve a field value, verify that the method supports the field by invoking PIMList.isSupportedField(int field).

A field can have an associated descriptive label to display to users. To retrieve a field label, invoke PIMList.getFieldLabel(int field).

Listeners

An application can implement the PIMListListener interface to receive notification when an item in a list changes. The PIMListListener interface provides three methods:

Method	Description
itemAdded(PIMItem item)	invoked when an item is added to a list
itemRemoved(PIMItem item)	invoked when an item is removed from a list
itemUpdated(PIMItem oldItem, PIMItem newItem)	invoked when an item changes

```
ContactList cl = (ContactList)PIM.getInstance().openPIMList(
   PIM.CONTACT_LIST, PIM.WRITE_ONLY);
((BlackBerryPIMList)cl).addListener(new PIMListListener() {
   public void itemAdded(PIMItem item) {
      System.out.println(" ITEM ADDED: " + ((Contact)item).getString(Contact.UID,
         0));
   }
   public void itemUpdated(PIMItem oldItem, PIMItem newItem) {
      System.out.println(" ITEM UPDATED: " +
      ((Contact)oldItem).getString(Contact.UID, 0) + " to " +
      ((Contact)newItem).getString(Contact.UID, 0));
   }
   public void itemRemoved(PIMItem item) {
      System.out.println(" ITEM REMOVED: " +
      ((Contact)item).getString(Contact.UID, 0));
   }
});
```

Note: The listener remains associated with the handheld database even after the corresponding PIMList object has been deleted. To remove the listener, invoke BlackBerryPIMList.removeListener().

Remote address lookup

To support remote address lookup, instantiate the BlackBerryContactList interface rather than the the ContactList interface. BlackBerryContactList contains the same functionality as ContactList, but provides additional methods to support remote address lookup.

The RemoteLookupListener interface provides a single method, items(), which returns an enumeration of the results of a remote address lookup.

Using the address book

Use an instance of `ContactList` to add or view contact information in the handheld address book. Create `Contact` objects to store individual contacts with information such as name, phone number, email address, and street address.

BlackBerry-specific address fields

The `BlackBerryContact` interface, which extends `Contact`, defines the following constants to provide access to fields that are specific to BlackBerry contacts:

- `BlackBerryContact.PIN:` provides access to the PIN field
- `BlackBerryContact.USER1` through `USER4:` provide access to the USER1 through USER4 fields

Invoke `BlackBerryPIMList.setFieldLabel()` to define labels for the USER1 through USER4 fields. The change takes effect immediately; you do not need to commit the change.

Note: Changing a label affects all contacts on the handheld.

Open a contact list

You must create a contact list before you can add contacts. Invoke `PIM.openPIMList()`. Provide as parameters the type of list to open (`PIM.CONTACT_LIST`) and the access mode with which to open the list (`READ_WRITE`, `READ_ONLY`, or `WRITE_ONLY`).

If you are writing an application specifically for BlackBerry handhelds, cast your contact list as a `BlackBerryContactList` because this interface provides additional methods to support remote address lookup. To make an application portable across multiple J2ME-compatible devices, use the PDAP implementation.

```
ContactList contactList = null;
try {
   contactList = (ContactList)PIM.getInstance().openPIMList(
   PIM.CONTACT_LIST, PIM.READ_WRITE);
} catch (PimException e) {
   return;
}
```

Create a contact

Invoke `createContact()` on a contact list.

```
Contact contact = contactList.createContact();
```

Note: The contact is not added to the database until you commit it. See " Save a contact" on page 27 for more information.

Add contact information

The `Contact` class defines fields in which to store data, such as `Contact.NAME`, `Contact.ADDR`, and `Contact.TEL`. Each field has a specific data type, which you can retrieve by invoking `PIMList.getFieldDataType(int field)`. Depending on the data type of the field, add a new value a by invoking one of the following methods: `addString()`, `addStringArray()`, `addDate()`, `addInt()`, `addBoolean()`, or `addBinary()`.

Before you set or retrieve a field, verify that the item supports the field by invoking `ContactList.isSupportedField(int field)`.

Some fields can store multiple values, using attributes to differentiate between values. For example, the `TEL` field supports the `ATTR_HOME`, `ATTR_WORK`, `ATTR_MOBILE`, and `ATTR_FAX` attributes to store numbers for work, home, mobile, and fax numbers. To determine how many values a field supports, invoke `PIMList.maxValues(int field)`. This method returns the number of values supported, or `-1` to indicate that an arbitrary number of values can be added. To verify that a field supports a particular attribute, invoke `isSupportedAttribute(int field, int attribute)`.

```
//create string array for name
try {
   ContactList contactList =
      (ContactList)PIM.getInstance().openPIMList(PIM.CONTACT_LIST,
      PIM.WRITE_ONLY);
} catch (PIMException e) {
}
Contact contact = contactList.createContact();
String[] name = new String[5]; // 5 name elements
try {
   name[Contact.NAME_PREFIX] = "Mr.";
   name[Contact.NAME_FAMILY] = "McPherson";
   name[Contact.NAME_GIVEN] = "Scott";
} catch (IllegalArgumentException iae) {
   // handle exception
}
//add name
if(contactList.isSupportedField(Contact.NAME)) {
   contact.addStringArray(Contact.NAME, Contact.ATTR_NONE, name);
}
//create string array for address
String[] address = new String[7]; // 7 address elements
try {
   address[Contact.ADDR_COUNTRY] = "United States";
   address[Contact.ADDR_LOCALITY] = "Los Angeles";
   address[Contact.ADDR_POSTALCODE] = "632300";
   address[Contact.ADDR_REGION] = "California";
   address[Contact.ADDR_STREET] = "323 Main Street";
} catch (IllegalArgumentException iae) {
   // handle exception
}
//add address
contact.addStringArray(Contact.ADDR, Contact.ATTR_NONE, address);
//add home telephone number
if (contactList.isSupportedField(Contact.TEL) &&
   contactList.isSupportedAttribute(Contact.TEL, Contact.ATTR_HOME)) {
   contact.addString(Contact.TEL, Contact.ATTR_HOME, "555-1234");
}
```

```
//add work telephone number
if (contactList.isSupportedField(Contact.TEL)) {
   contact.addString(Contact.TEL, Contact.ATTR_HOME, "555-5555");
}
//add work email address
if (contactList.isSupportedField(Contact.EMAIL)) {
   contact.addString(Contact.EMAIL, Contact.ATTR_WORK,
   "scott.mcpherson@blackberry.com");
}
```

Modify contact information

For fields that support only one value, invoke the appropriate `set` method to replace an existing value with a new value.

Note: If you invoke an `add` method, such as `addString()`, for a field that already has a value, a `FieldFullException` is thrown. Use the corresponding `set` method, such as `setString()`, to change an existing value.

For the name and address fields, which contain a string array value, retrieve the array and then modify one or more indexes in the array before adding it back in.

```
if (contact.countValues(Contact.NAME) > 0) {
   String[] newname = contact.getStringArray(Contact.NAME, 0);
}
// Change the prefix to Dr. and add the suffix, Jr.
newname[Contact.NAME_PREFIX] = "Dr.";
newname[Contact.NAME_SUFFIX] = "Jr.";
contact.setStringArray(Contact.NAME, 0, Contact.ATTR_NONE, newname);
```

For fields that support multiple values, verify that the maximum number of values is not exceeded before adding another value.

```
if (contact.countValues(Contact.EMAIL) < contactList.maxValues(Contact.EMAIL)) {
   contact.addString(Contact.EMAIL, Contact.ATTR_NONE,
      "scott.mcpherson@blackberry.com");
}
```

Save a contact

Invoke `commit()`. Before you commit the change, invoke `isModified()` to determine whether any contact fields have changed since the contact was last saved.

```
if(contact.isModified()) {
   contact.commit();
}
```

Retrieve contact information

Invoke `PIMList.items()`.

Note: When you invoke `PIMList.items()` to retrieve an enumeration of items in a list, the order of items is undefined. Your application must sort items as necessary.

For a particular contact, invoke `PIMItem.getFields()` to retrieve an array of IDs for fields that have data. Invoke `PIMItem.getString()` to retrieve the field values.

```
ContactList contactList = (ContactList)PIM.getInstance().openPIMList(
PIM.CONTACT_LIST, PIM.READ_WRITE);
Enumeration enum = contactList.items();
while (enum.hasMoreElements()) {
   Contact c = (Contact)enum.nextElement();
   int[] fieldIds = c.getFields();
   int id;
   for(int index = 0; index < fieldIds.length; ++index) {
      id = fieldIds[index];
      if(c.getPIMList().getFieldDataType(id) == Contact.STRING) {
         for(int j=0; j < c.countValues(id); ++j) {
            String value = c.getString(id, j);
            System.out.println(c.getPIMList().getFieldLabel(id) + "=" + value);
         }
      }
   }
}
```

Convert a contact to a serial format

To import or export PIM item data, use an output stream writer to export tasks from the handheld to a supported serial format, such as iCal and vCard.

To retrieve a string array of supported formats, invoke `PIM.supportedSerialFormats()` and specify the list type (`PIM.Contact_LIST`).

To write an item to a supported serial format, invoke `toSerialFormat()`.

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE." This parameter cannot be null.

```
ContactList contactList = (ContactList)PIM.getInstance().openPIMList(
   PIM.CONTACT_LIST, PIM.READ_ONLY);
String[] dataFormats = PIM.getInstance().supportedSerialFormats(
   PIM.CONTACT_LIST);
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
Enumeration e = contactList.items();
while (e.hasMoreElements()) {
   Contact c = (Contact)e.nextElement();
   PIM.getInstance().toSerialFormat(c, byteStream, "UTF8", dataFormats[0]);
}
```

Import a contact

Invoke `fromSerialFormat()`, which returns an array of PIM items. To create a new contact using the PIM item, invoke `ContactList.importContact()`

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE." This parameter cannot be null.

```
//import contact from vCard
ByteArrayInputStream is = new ByteArrayInputStream(outputStream.toByteArray());
```

```
PIMItem[] pi = PIM.getInstance().fromSerialFormat(istream, "UTF8");
ContactList contactList =
   (ContactList)PIM.getInstance().openPIMList(PIM.CONTACT_LIST, PIM.READ_WRITE);
Contact contact2 = contactList.importContact((Contact)pi[0]);
contact2.commit();
```

Delete a contact

Invoke `removeContact()` on a contact list.

```
contactList.removeContact(contact);
```

Close a contact list

Invoke `close()`.

```
try {
   contactList.close();
} catch(PIMException e) {
   Dialog.alert(e.toString());
}
```

Code example

The following example demonstrates how to display a screen that enables users to add new contacts to the handheld address book. After you save a contact, open the address book to verify that the contact was saved.

Example: ContactsDemo.java

```
/**
 * ContactsDemo.java
 * Copyright (C) 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.contactsdemo;

import java.io.*;
import java.util.*;
import javax.microedition.pim.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.i18n.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import com.rim.samples.docs.baseapp.*;
import net.rim.blackberry.api.pdap.*;

public final class ContactsDemo extends BaseApp {
    private ContactScreen _contactScreen;
    public static void main(String[] args) {
```

```
        new ContactsDemo().enterEventDispatcher();
    }
    public ContactsDemo() {
        _contactScreen = new ContactScreen();
        pushScreen(_contactScreen);
    }
    protected void onExit() {
    }
    // Inner class. Creates a Screen to add a contact.
    public static final class ContactScreen extends MainScreen {
        private EditField _first, _last, _email, _phone, _pin;
        private SaveMenuItem _saveMenuItem;
        private class SaveMenuItem extends MenuItem {
            private SaveMenuItem() {
                super(null, 0, 100000, 5);
            }
            public String toString() {
                return "Save";
            }
            public void run() {
                onSave();
            }
        }
        public ContactScreen() {
            _saveMenuItem = new SaveMenuItem();
            setTitle(new LabelField("Contacts Demo", LabelField.ELLIPSIS |
LabelField.USE_ALL_WIDTH));
            _first = new EditField("First Name: ", "");
            add(_first);
            _last = new EditField("Last Name: ", "");
            add(_last);
            _email = new EditField("Email Address: ", "",
            BasicEditField.DEFAULT_MAXCHARS, BasicEditField.FILTER_EMAIL);
            add(_email);
            _phone = new EditField("Work Phone: ", "",
            BasicEditField.DEFAULT_MAXCHARS, BasicEditField.FILTER_PHONE);
            add(_phone);
           _pin = new EditField("PIN:", "", 8, BasicEditField.FILTER_HEXADECIMAL);
            add(_pin);
        }
        protected boolean onSave() {
            String firstName = _first.getText();
            String lastName = _last.getText();
            String email = _email.getText();
            String phone = _phone.getText();
            String pin = _pin.getText();
            // Verify that a first or last name and email has been entered.
           if ((firstName.equals("") && lastName.equals("")) || email.equals("")) {
                Dialog.inform("You must enter a name and an email address!");
                return false;
            } else {
                try {
                    ContactList contactList =
(ContactList)PIM.getInstance().openPIMList(PIM.CONTACT_LIST, PIM.WRITE_ONLY);
                    Contact contact = contactList.createContact();
```

```
                    String[] name = new
    String[contactList.stringArraySize(Contact.NAME)];
                    // Add values to PIM item.
                    if (!firstName.equals("")) {
                        name[Contact.NAME_GIVEN] = firstName;
                    }
                    if (!lastName.equals("")) {
                        name[Contact.NAME_FAMILY] = lastName;
                    }
                    contact.addStringArray(Contact.NAME, Contact.ATTR_NONE, name);
                    contact.addString(Contact.EMAIL, Contact.ATTR_HOME, email);
                    contact.addString(Contact.TEL, Contact.ATTR_WORK, phone);
                    if (contactList.isSupportedField(BlackBerryContact.PIN)) {
                      contact.addString(BlackBerryContact.PIN, Contact.ATTR_NONE,
    pin);
                    }
                    // Save data to address book.
                    contact.commit();
                    // Reset UI fields.
                    _first.setText("");
                    _last.setText("");
                    _email.setText("");
                    _phone.setText("");
                    _pin.setText("");
                    return true;
                } catch (PIMException e) {
                    return false;
                }
            }
        }
        protected void makeMenu(Menu menu, int instance) {
            menu.add(_saveMenuItem);
            super.makeMenu(menu, instance);
        }
    }
}
```

Using tasks

Use an instance of the `ToDoList` class to store a list of tasks, or to do items. Create one or more `ToDo` objects to store data for each task, such as summary, priority, due date, and status.

Open a task list

Invoke `PIM.openPIMList()`. Provide as parameters the type of list to open (`PIM.TODO_LIST`) and the access mode with which to open the list (`READ_WRITE`, `READ_ONLY`, or `WRITE_ONLY`).

```
ToDoList todoList = null;
try {
    todoList = (ToDoList)PIM.getInstance().openPIMList(
PIM.TODO_LIST, PIM.READ_WRITE);
} catch (PimException e) {
```

```
    //an error occurred
    return;
}
```

Create a task

Invoke `createToDo()` on a task list.

```
ToDo task = todoList.createToDo();
```

Note: The task is not added to the database until you commit it. See " Save a task" on page 33 for more information.

Add task information

The `ToDo` class defines fields in which to store data, such as `SUMMARY`, `PRIORITY`, and `DUE`. Each field has a specific data type, which you can retrieve by invoking `PIMList.getFieldDataType(int field)`. Depending on the data type of the field, set the field data by invoking one of the following methods: `addString()`, `addDate()`, `addInt()`, `addBoolean()`, or `addBinary()`.

See " PIM APIs" on page 23 for more information on fields.

Before you set or retrieve a field, verify that the item supports the field by invoking `isSupportedField( int field )`.

```
if (task.isSupportedField(ToDo.SUMMARY)) {
    task.addString(ToDo.SUMMARY, ToDo.ATTR_NONE, "Create project plan");
}
if (task.isSupportedField(ToDo.DUE)) {
    Date date = new Date();
    task.addDate(ToDo.DUE, ToDo.ATTR_NONE, (date + 17280000));
}
if (task.isSupportedField(ToDo.NOTE)) {
    task.addString(ToDo.NOTE, ToDo.ATTR_NONE, "Required for meeting");
}
if (task.isSupportedField(ToDo.PRIORITY)) {
    task.addInt(Todo.PRIcORITY, ToDo.ATTR_NONE, 2);
}
```

Set the status of a task

Use the PIM extended field `ToDo.EXTENDED_FIELD_MIN_VALUE + 9`.

Constant	Value
STATUS_NOT_STARTED	1
STATUS_IN_PROGRESS	2
STATUS_COMPLETED	3
STATUS_WAITING	4
STATUS_DEFERRED	5

```
task.addInt(ToDo.EXTENDED_FIELD_MIN_VALUE + 9, ToDo.ATTR_NONE, 2);
```

Modify task information

Invoke the appropriate `set` method, such as `setString()`, to replace an existing value with a new value. Invoke `countValues()` to determine if a value is already set for the field.

Note: If you invoke an `add` method, such as `addString()`, when a value already exists, a `FieldFullException` is thrown. Use the corresponding `set` method, such as `setString()`, to change an existing value.

```
if (task.countValues(ToDo.SUMMARY) > 0) {
   task.setString(ToDo.SUMMARY, 0, ToDo.ATTR_NONE, "Review notes");
}
```

Save a task

Invoke `commit()`. Before you save, invoke `isModified()` to determine whether any task fields have changed since the task was last saved.

```
if(task.isModified()) {
   task.commit();
}
```

Retrieve task information

Retrieve an enumeration

Invoke `PIMList.items()` on the task list.

```
ToDoList todoList = (ToDoList)PIM.getInstance().openToDoList(
   PIM.TODO_LIST, PIM.READ_ONLY);
Enumeration enum = todoList.items();
```

Retrieve field IDs and values for a task

To retrieve an array of IDs for fields that have data for a particular `ToDo` item, invoke `PIMItem.getFields()`. To retrieve the field values, invoke `PIMItem.getString()`.

```
while (enum.hasMoreElements()) {
   ToDo task = (ToDo)enum.nextElement();
   int[] fieldIds = task.getFields();
   int id;
   for(int index = 0; index < fieldIds.length; ++index) {
      id = fieldIds[index];
      if(task.getPIMList().getFieldDataType(id) == STRING) {
         for(int j=0; j < task.countValues(id); ++j) {
            String value = task.getString(id, j);
            System.out.println(task.getFieldLable(id) + "=" + value);
         }
      }
   }
}
```

Convert a task to a to serial format

To import or export PIM item data, use an output stream writer to export tasks from the handheld to a supported serial format, such as iCal and vCard.

To retrieve a string array of supported serial formats, invoke `PIM.supportedSerialFormats()` and specify the list type (`PIM.TODO_List`).

To write an item to a serial format, invoke `toSerialFormat()`.

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE." This parameter cannot be null.

```
ToDoList todoList = (ToDoList)PIM.getInstance().openPIMList(
PIM.TODO_LIST, PIM.READ_ONLY);
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
String[] dataFormats = PIM.getInstance().supportedSerialFormats(PIM.TODO_LIST);
Enumeration e = todoList.items();
while (e.hasMoreElements()) {
   ToDo task = (ToDo)e.nextElement();
   PIM.getInstance().toSerialFormat(task, byteStream, "UTF8", dataFormats[0]);
}
```

Import a task

Invoke `fromSerialFormat()`, which returns an array of `PIMItem` objects. To create a new task using the PIM items, invoke `ToDoList.importToDo()`.

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE." This parameter cannot be null.

```
String[] dataFormats = PIM.toDoSerialFormats();
//write task to vCard
ByteArrayOutputStream os = new ByteArrayOutputStream();
PIM.getInstance().toSerialFormat(task, os, "UTF8", dataFormats[0]);
//import task from vCard
ByteArrayInputStream is = new ByteArrayInputStream(outputStream.toByteArray());
PIMItem[] pi = PIM.getInstance().fromSerialFormat(is, "UTF8");
ToDoList todoList = (ToDoList)PIM.getInstance().openPIMList(
   PIM.TODO_LIST, PIM.READ_WRITE);
ToDo task2 = todoList.importToDo((ToDo)pi[0]);
```

Tip: The `importToDo()` method saves the task, so you do not have to invoke `commit()`.

Delete a task

Invoke `removeToDo()` on a task list.

```
todoList.removeToDo(task);
```

Close a task list

Invoke `todoList.close()`. Make sure you catch applicable exceptions.

```
try {
```

```
    todoList.close();
} catch (PimException e) {
    // handle exception
}
```

Code example

Example: TaskDemo.java

```
/**
 * TaskDemo.java
 * Copyright (C) 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.taskdemo;
import java.io.*;
import java.util.*;
import javax.microedition.pim.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.i18n.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import com.rim.samples.docs.baseapp.*;

public final class TaskDemo extends BaseApp {
    private TaskScreen _taskScreen;
    public static void main(String[] args) {
        new TaskDemo().enterEventDispatcher();
    }
    private TaskDemo() {
        _taskScreen = new TaskScreen();
        pushScreen(_taskScreen);
    }
    protected void onExit() {
    }
    public final static class TaskScreen extends MainScreen {
        // Members.
        private EditField _summary, _note;
        private DateField _due;
        private ObjectChoiceField _priority, _status;
        private SaveMenuItem _saveMenuItem;
        private class SaveMenuItem extends MenuItem {
            private SaveMenuItem() {
                super(null, 0, 100000, 5);
            }
            public String toString() {
                return "Save";
            }
            public void run() {
                onSave();
            }
        }
```

```
        public TaskScreen() {
            _saveMenuItem = new SaveMenuItem();
            setTitle(new LabelField("Tasks Demo",
            LabelField.ELLIPSIS | LabelField.USE_ALL_WIDTH));
            _summary = new EditField("Task Summary: ", "");
            add(_summary);
            // In TODO.Priority, 0 to 9 is highest to lowest priority.
            String[] choices = {"High", "Normal", "Low"};
            _priority = new ObjectChoiceField("Priority: ", choices, 1);
            add(_priority);
            String[] status = { "Not Started", "In Progress", "Completed",
            "Waiting on someone else", "Deferred" };
            _status = new ObjectChoiceField("Status: ", status, 0);
            add(_status);
            _due = new DateField("Due: ", System.currentTimeMillis() + 3600000,
            DateField.DATE_TIME);
            add(_due);
            _note = new EditField("Extra Notes: ", "");
            add(_note);
        }
        protected boolean onSave() {
            try {
                ToDoList todoList = (ToDoList)PIM.getInstance().
                openPIMList(PIM.TODO_LIST, PIM.WRITE_ONLY);
                ToDo task = todoList.createToDo();
                task.addDate(ToDo.DUE, ToDo.ATTR_NONE, _due.getDate());
                task.addString(ToDo.SUMMARY, ToDo.ATTR_NONE, _summary.getText());
                task.addString(ToDo.NOTE, ToDo.ATTR_NONE, _note.getText());
                task.addInt(ToDo.PRIORITY, ToDo.ATTR_NONE,
                _priority.getSelectedIndex());
                // ToDo.EXTENDED_FIELD_MIN_VALUE + 9 represents status.
                // Add 1 to selected index so that values are correct.
                // See the RIM Implementation Notes in the API docmentation for
   ToDo.
                task.addInt(ToDo.EXTENDED_FIELD_MIN_VALUE + 9, ToDo.ATTR_NONE,
                _status.getSelectedIndex() + 1);
                // Save task to handheld tasks.
                task.commit();
                _summary.setText("");
                _note.setText("");
                _due.setDate(null);
                _priority.setSelectedIndex(1); // Reset to ìNormalî priority.
                _status.setSelectedIndex(0); // Reset to ìNot Startedî status.
                return true;
            } catch (PIMException e) {
                return false;
            }
        }
        protected void makeMenu(Menu menu, int instance) {
            menu.add(_saveMenuItem);
            super.makeMenu(menu, instance);
        }
    }
}
```

Using the calendar

Use an instance of the `EventList` class to access the handheld calendar. Create one or more `Event` objects to store information for specific appointments. For each event, you can store data such as the summary, location, start and end time, and reminder notification.

Open an event list

You must create an `EventList` before you can add events. Invoke `PIM.openPIMList()`. Provide as parameters the type of list to open (`PIM.EVENT_LIST`) and the mode in which to open the list (`READ_WRITE`, `READ_ONLY`, or `WRITE_ONLY`).

```
EventList eventList = null;
try {
   eventList = (EventList)PIM.getInstance().openPIMList(
   PIM.EVENT_LIST, PIM.READ_WRITE);
} catch (PimException e) {
   // handle exception
}
```

Create an appointment

Invoke `createEvent()` on an event list.

```
Event event = eventList.createEvent();
```

Note: The appointment is not added to the database until you commit the change. See " Save an appointment" on page 38 for more information.

Add appointment information

The `Event` class defines fields in which to store data, such as `SUMMARY`, `LOCATION`, `START`, `END`, and `ALARM`. Each field has a specific data type, which you can retrieve by invoking `PIMList.getFieldDataType(int field)`. Depending on the data type of the field, set the field data by invoking one of the following methods: `addString()`, `addDate()`, `addInt()`, `addBoolean()`, or `addBinary()`.

Before you set or retrieve a field, verify that the item supports the field by invoking `isSupportedField( int field )`.

```
if (event.isSupportedField(Event.SUMMARY)) {
   event.addString(Event.SUMMARY, Event.ATTR_NONE, "Meet with customer");
}
if (event.isSupportedField(Event.LOCATION)) {
   event.addString(Event.LOCATION, Event.ATTR_NONE, "Conference Center");
}
Date start = new Date(System.currentTimeMillis() + 8640000);
if (event.isSupportedField(Event.START)) {
   event.addDate(Event.START, Event.ATTR_NONE, start);
}
if (event.isSupportedField(Event.END)) {
   event.addDate(Event.END, Event.ATTR_NONE, start + 72000000);
}
```

```
if (event.isSupportedField(Event.ALARM)) {
   if (event.countValues(Event.ALARM) > 0) {
      eventValue(Event.ALARM,0);
      event.setInt(Event.ALARM, 0, Event.ATTR_NONE, 396000);
   }
}
```

Tip: If the `Event.ALARM` field is not set, the appointment reminder is automatically set to 15 minutes before the start of the event.

Create a recurring appointment

To define a recurrence pattern for an appointment, use a `RepeatRule` object. The `RepeatRule` class defines fields for the properties and values that you can set, such as `COUNT`, `FREQUENCY`, and `INTERVAL`. To retrieve an array of supported fields, invoke `RepeatRule.getFields()`.

Define a recurrence pattern

Invoke `setInt()` or `setDate()` on a new `RepeatRule` object.

```
RepeatRule recurring = new RepeatRule();
recurring.setInt(RepeatRule.FREQUENCY, RepeatRule.MONTHLY);
recurring.setInt(RepeatRule.DAY_IN_MONTH, 14);
```

Assign a recurrence pattern to an appointment

Invoke `setRepeat()` on an event.

```
EventList eventList = (EventList)PIM.getInstance().openPIMList(
PIM.EVENT_LIST, PIM.READ_WRITE);
Event event = eventList.createEvent();
event.setRepeat(recurring);
```

Modify appointment information

To replace an existing value with a new one, invoke the appropriate set method, such as `setString()`. To determine if a value is already set for the field, invoke `countValues()`.

Note: If you invoke an `add` method, such as `addString()`, when a value already exists, a `FieldFullException` is thrown. Use the corresponding `set` method, such as `setString()`, to change an existing value.

```
if (event.countValues(Event.LOCATION) > 0) {
   event.setString(Event.LOCATION, 0, Event.ATTR_NONE, "Board Room");
}
```

Save an appointment

Tip: The `importEvent()` method saves the appointment, so you do not have to invoke `commit()`.

Invoke `commit()`. Before you save the appointment, invoke `isModified()` to determine whether any of the appointment fields have changed since the appointment was last saved.

```
if(event.isModified()) {
   event.commit();
```

```
}
```

Retrieve appointment information

Retrieve an enumeration of appointments

Invoke `PIMList.items()`.

```
EventList eventList = (EventList)PIM.getInstance().openPIMList(
PIM.EVENT_LIST, PIM.READ_ONLY);
Enumeration e = eventList.items();
```

Retrieve field IDs and values

To retrieve an array of IDs of fields that have data for a particular task, invoke `PIMItem.getFields()`. To retrieve the field values, invoke `PIMItem.getString()`.

```
while (e.hasMoreElements()) {
   Event event = (Event)e.nextElement();
   int[] fieldIds = event.getFields();
   int id;
   for(int index = 0; index < fieldIds.length; ++index) {
      id = fieldIds[index];
      if(e.getPIMList().getFieldDataType(id) == STRING) {
         for(int j=0; j < event.countValues(id); ++j) {
            String value = event.getString(id, j);
            System.out.println(event.getFieldLable(id) + "=" + value);
         }
      }
   }
}
```

Convert an appointment to a serial format

To import or export PIM item data, use an output stream writer to export tasks from the handheld to a supported serial format, such as iCal and vCard.

To retrieve a string array of supported serial formats, invoke `PIM.supportedSerialFormats()` and specify the list type (`PIM.EVENT_List`).

To write an item to a serial format, invoke `toSerialFormat()`.

```
EventList eventList = (EventList)PIM.getInstance().openPIMList( PIM.EVENT_LIST,
   PIM.READ_ONLY );
ByteArrayOutputStream bytestream = new ByteArrayOutputStream();
```

To write an item to a serial format, invoke `toSerialFormat()`.

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE" . This parameter cannot be null.

```
Enumeration e = eventList.items();
while (e.hasMoreElements()) {
   Event event = (Event)e.nextElement();
   PIM.getInstance().toSerialFormat(event, byteStream, "UTF8", dataFormats[0]);
```

```
}
```

Import an appointment

Invoke `fromSerialFormat(java.io.InputStream is, java.lang.String enc)`, which returns an array of `PIMItem` objects. Invoke `EventList.importEvent()` to add a new appointment.

Note: The `enc` parameter specifies the character encoding to use when writing to the output stream. Supported character encodings include " UTF8," " ISO-8859-1," and " UTF-16BE." This parameter cannot be null.

```
// Convert an existing appointment into a vCard and then import the vCard as a new
// appointment
String[] dataFormats = PIM.eventSerialFormats();
//write appointment to vCard
ByteArrayOutputStream os = new ByteArrayOutputStream();
PIM.getInstance().toSerialFormat(event, os, "UTF8", dataFormats[0]);
//import appointment from vCard
ByteArrayInputStream is = new ByteArrayInputStream(outputStream.toByteArray());
PIMItem[] pi = PIM.getInstance().fromSerialFormat(is, "UTF8");
EventList eventList = (EventList)PIM.getInstance().openPIMList(
PIM.EVENT_LIST, PIM.READ_WRITE);
Event event2 = eventList.importEvent((Event)pi[0]);
```

Close an event list

Invoke `close()`.

```
try {
   eventList.close();
} catch (PimException e) {
   // handle exception
}
```

Code example

The following example displays a screen that enables users to create new, recurring appointments in the handheld calendar. You could combine this sample with ContactsDemo.java to allow the user to invite attendees to the meeting. See "ContactsDemo.java" on page 29 for more information.

After you save an appointment, click the **Calendar** icon to verify that the appointment was saved.

Example: EventDemo.java

```
/**
 * EventDemo.java
 * Copyright (C) 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.eventdemo;
import java.io.*;
import java.util.*;
import javax.microedition.pim.*;
import net.rim.device.api.ui.*;
```

```
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.i18n.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import com.rim.samples.docs.baseapp.*;

public final class EventDemo extends BaseApp {
    private EventScreen _eventScreen;
    public static void main(String[] args)  {
        new EventDemo().enterEventDispatcher();
    }
    private EventDemo()     {
        _eventScreen = new EventScreen();
        pushScreen(_eventScreen);
    }
    protected void onExit() {
    }
    public final static class EventScreen extends MainScreen {
        private EditField _subject, _location;
        private SaveMenuItem _saveMenuItem;
        private DateField _startTime, _endTime;
        private ObjectChoiceField _repeat;
        private Event event;
        private class SaveMenuItem extends MenuItem {
            public SaveMenuItem() {
                super(null, 0, 100000, 5);
            }
            public String toString() {
                return "Save";
            }
            public void run() {
                onSave();
            }
        }
        public EventScreen() {
            _saveMenuItem = new SaveMenuItem();
           setTitle(new LabelField("Event Demo", LabelField.ELLIPSIS |
                LabelField.USE_ALL_WIDTH) );
            _subject = new EditField("Subject: ", "");
            add(_subject);
            _location = new EditField("Location: ", "");
            add(_location);
            _startTime = new DateField("Start: ", System.currentTimeMillis() +
                3600000, DateField.DATE_TIME);
            _endTime = new DateField("End: ", System.currentTimeMillis() +
                7200000, DateField.DATE_TIME);
            add(new SeparatorField());
            add(_startTime);
            add(_endTime);
            add(new SeparatorField());
            String[] choices = {"None", "Daily", "Weekly", "Monthly", "Yearly"};
            _repeat = new ObjectChoiceField("Recurrence: ", choices, 0);
            add(_repeat);
        }
```

```
        protected boolean onSave() {
            try {
                EventList eventList = (EventList)PIM.getInstance().
                    openPIMList(PIM.EVENT_LIST, PIM.WRITE_ONLY);
                event = eventList.createEvent();
                event.addString(Event.SUMMARY, PIMItem.ATTR_NONE,
                    _subject.getText());
                event.addString(Event.LOCATION, PIMItem.ATTR_NONE,
                    _location.getText());
                event.addDate(Event.END, PIMItem.ATTR_NONE, _endTime.getDate());
                event.addDate(Event.START, PIMItem.ATTR_NONE,
                    _startTime.getDate());
                if(_repeat.getSelectedIndex() != 0) {
                    event.setRepeat(setRule());
                }
                // Save the appointment to the Calendar.
                event.commit();
                //reset fields on screen
                _subject.setText("");
                _location.setText("");
                _endTime.setDate(null);
                _startTime.setDate(null);
                _repeat.setSelectedIndex(0);
                return true;
            } catch (PIMException e) {
                System.err.println(e);
            }
            return false;
        }
        private RepeatRule setRule() {
            RepeatRule rule = new RepeatRule();
            int index = _repeat.getSelectedIndex();
                if (index == 0) {
                    rule.setInt(RepeatRule.FREQUENCY, RepeatRule.DAILY);
                }
                if (index == 1) {
                    rule.setInt(RepeatRule.FREQUENCY, RepeatRule.WEEKLY);
                }
                if (index == 2) {
                    rule.setInt(RepeatRule.FREQUENCY, RepeatRule.MONTHLY);
                }
                if (index == 3) {
                    rule.setInt(RepeatRule.FREQUENCY, RepeatRule.YEARLY);
                }
                return rule;
        }
        protected void makeMenu(Menu menu, int instance)                {
            menu.add(_saveMenuItem);
            menu.addSeparator();
            super.makeMenu(menu, instance);
        }
    }
}
```

4

Adding handheld options

- Options API
- Adding option items

Options API

The BlackBerry options API, in the `net.rim.blackberry.api.options` package, enables you to add items to the handheld options. Use this capability to add system-wide options to the handheld that multiple applications can use.

When users click the **Options** icon on the handheld Home screen, a list of options, such as AutoText, Date/Time, and Firewall, appears. The user can select one of these items to view a screen for that particular option. The screen displays one or more fields. Typically, the user can change the value of each field.

Adding option items

Registering to add options

Implement the `OptionsProvider` interface, including the `getTitle()`, `save()`, and `populateMainScreen()` methods.

Create a library project

To add option items when the handheld starts, create a library project with a `libMain()` method to perform the required registration.

1. In the IDE, create a project.
2. Right-click the project and click **Properties**.
3. In the Properties window, click the **Application** tab.
4. In the **Project type** drop-down list, click **Library**.
5. Select the **Auto-run on startup** option.
6. Click **OK**.

Register as an options provider

Implement `getInstance()` to retrieve a static instance of your class. Only one instance should exist at a time.

Invoke `registerOptionsProvider()` in `libMain()`. Provide as parameter a static instance of your class.

```
private static DemoOptionsProvider _instance;
...
public static DemoOptionsProvider getInstance() {
```

```
    if(_instance == null) {
        _instance = new DemoOptionsProvider("Options Demo");
    }
    return _instance;
}

...
public static void libMain(String[] args) {
    OptionsManager.registerOptionsProvider(getInstance());
}
```

Store options

To store the option value that is currently selected, implement the `Persistable` interface. In your implementation, define methods for setting the selected option value, and committing and retrieving an option value in the persistent store.

Note: If you implement the `Persistable` interface as an inner class, make it—and its `get()`, `set()`, and `commit()` methods—public so that other applications can access your options data.

See " Managing persistent data" on page 81 for more information on storing persistent data.

Provide access to option data

In your library class, add public methods to enable other applications to access your option data.

Code example

Example: DemoOptionsProvider.java

```
/**
 * DemoOptionsProvider.java
 * Copyright 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.demooptionsprovider;
import net.rim.blackberry.api.options.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.i18n.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;

// A simple library class to demonstrate the use of the options facilities.
public final class DemoOptionsProvider implements OptionsProvider {
    // members
    private ObjectChoiceField _ocf;
    private OptionsDemoData _data;
    private String _title;
    private static DemoOptionsProvider _instance;
    // constructors
    private DemoOptionsProvider()   {
```

```
}
private DemoOptionsProvider(String title)       {
    _title = title;
    _data = OptionsDemoData.load();
}
// Only allow one instance of this class.
public static DemoOptionsProvider getInstance() {
    if (_instance == null) {
        _instance = new DemoOptionsProvider("Options Demo");
    }
    return _instance;
}
// On startup, create the instance and register it.
public static void libMain(String[] args)       {
    OptionsManager.registerOptionsProvider(getInstance());
}
// Get the title for the option item.
public String getTitle() {
    return _title;
}
// Add fields to the screen.
public void populateMainScreen(MainScreen screen) {
    int index = _data.getSelected();
    String[] choices = {"High", "Low", "None"};
    _ocf = new ObjectChoiceField("Security: ", choices, index);
    screen.add(_ocf);
}
// Save the data.
public void save()      {
    _data.setSelected(_ocf.getSelectedIndex());
    _data.commit();
}
// Retrieve the data. Used by other applications to access options data.
public OptionsDemoData getData() {
    return _data;
}
// Inner class to store selected option values.
public static final class OptionsDemoData implements Persistable        {
    private static final long ID = 0x6af0b5eb44dc5164L;
    private int _selectedOption;
    private OptionsDemoData() {
    }
    public int getSelected() {
        return _selectedOption;
    }
    public void setSelected(int index) {
        _selectedOption = index;
    }
    public void commit() {
        PersistentObject.commit(this);
    }
    private static OptionsDemoData load() {
        PersistentObject persist = PersistentStore.getPersistentObject(
            OptionsDemoData.ID );
        OptionsDemoData contents = (OptionsDemoData)persist.getContents();
```

```
            synchronized( persist ) {
                if( contents == null ) {
                    contents = new OptionsDemoData();
                    persist.setContents( contents );
                    persist.commit();
                }
            }
            return contents;
        }
    }
}
```

5

BlackBerry Browser

- Browser APIs
- Displaying web content
- Supporting additional MIME types
- Registering as a HTTP filter

Browser APIs

API name and package	Description
Browser application API (net.rim.blackberry.api.browser)	This API enables applications to display web content, including supported image types, HTML and WML pages, by invoking the BlackBerry Browser. It also enables applications to supply a referrer, HTTP headers and post data in an HTTP request.
Browser field API (net.rim.blackberry.api.browser.field)	This API enables an application to retrieve web content for display in a browser field, which is included in the application UI. This API also enables applications to configure the appearance of the browser field, such as by eliminating the scroll bar or specifying displaying the browser field in only a portion of the screen.
Browser page API (net.rim.blackberry.api.browser.plugin)	This API enables applications to add support for additional MIME types. By registering as rendering provider for a MIME type when the handheld starts, all subsequent browser sessions will support the additional MIME type.
HTTP Filter API (net.rim.device.api.io.http)	This API enables applications to register with the browser as provider for one or more URLs.

Displaying web content

Displaying web content in the browser

To display web content in the BlackBerry Browser, use the browser application API (`net.rim.blackberry.api.browser`).

Retrieve a browser session

To retrieve the default `BrowserSession` object, invoke the static method `Browser.getDefaultSession()`. This object gives you access to the running browser on the handheld.

Note: Retrieving the default session overrides any open sessions on the handheld.

To retrieve a different session, invoke `getSession()`. This method retrieves a browser configuration service record according to its unique ID (UID). See " Service book API" on page 103 for more information.

Request a web page

To request a web page, invoke `BrowserSession.displayPage()`. The example below uses the version of `displayPage()` that accepts only a URL. To specify a referrer, HTTP headers, and post data, use the version that accepts these additional parameters.

Code sample

The following excerpt from the Restaurants.java sample creates a menu item that displays a web page in the browser.

```
private MenuItem browserItem = new
    MenuItem(_resources.getString(MENUITEM_BROWSER), 110, 12) {
    public void run() {
        synchronized(store) {
            String websiteUrl = websitefield.getText();
            if (websiteUrl.length == 0) {
                Dialog.alert(_resources.getString(ALERT_NO_WEBSITE));
            } else {
                BrowserSession visit = Browser.getDefaultSession();
                visit.displayPage(websiteUrl);
            }
        }
    }
};
```

Displaying web content in a browser field

To include a browser field within an application UI, use the browser field API (`net.rim.device.api.browser.field`). The browser rendering library handles the rendering of web content for the field, and then returns a `BrowserField`, a field in which URL content is rendered, to your application for display.

Note: The browser session that is used to open a browser field is independent of the default browser session on the handheld. Any open browser sessions are unaffected.

The `RenderingApplication` interface defines the callback functionality a rendering session requires to assist with handling URL resources. To display web content in a browser field, implement the `RenderingApplication` interface.

Create a separate thread for rendering

To prevent the application from hanging while the application retrieves and displays the browser field, perform these actions on a separate thread.

```
class CreationThread extends Thread {
    BrowserFieldHandlerApplication _callBackApplication;
    BasicRenderingApplication  _renderingApplication;
    public CreationThread(BrowserFieldHandlerApplication callBackApplication) {
        _callBackApplication = callBackApplication;
    }
    public void run() {
        _renderingApplication = new
    BasicRenderingApplication(_callBackApplication);
        BrowserField field = _renderingApplication.getBrowserField("www.rim.com");
        _callBackApplication.displayBrowserField(field);
```

```
    }
}
```

Set rendering options

Override `getRenderingOptions()`. If you do not override this method, default rendering options are used. See `RenderingOptions` in the *API Reference* for more information.

Handle events

To handle events, such as a URL request, implement `eventOccurred()`.

```
public Object eventOccurred(Event event) {
    int eventId = event.getUID();
    switch (eventId) {
        case Event.EVENT_URL_REQUESTED : {
            UrlRequestedEvent e = (UrlRequestedEvent) event;
            // this is a regular request
            String absoluteUrl = e.getURL();
            HttpConnection conn = null;
            OutputStream out = null;
            try {
                conn = (HttpConnection) Connector.open(absoluteUrl);
                    FormData postData = e.getPostData();
                    if (postData == null) {
                        conn.setRequestMethod(HttpConnection.GET);
                    } else {
                        conn.setRequestMethod(HttpConnection.POST);
                        byte[] postBytes = postData.getBytes();
                        conn.setRequestProperty(
                            HttpProtocolConstants.HEADER_CONTENT_LENGTH,
                            String.valueOf(postBytes.length));
                        if (conn.getRequestProperty(
                            HttpProtocolConstants.HEADER_CONTENT_TYPE) == null) {
                            conn.setRequestProperty(
                                HttpProtocolConstants.HEADER_CONTENT_TYPE,
                            postData.getContentType());
                        }
                        out = conn.openOutputStream();
                        out.write(postBytes);
                    }
                    HttpHeaders requestHeaders = e.getHeaders();
                    if (requestHeaders != null) {
                        /* From
                        http://www.w3.org/Protocols/rfc2616/rfc2616-
                        sec15.html#sec15.1.3
                        ...
                        Clients SHOULD NOT include a Referer header field in a
                        (non-secure) HTTP request if the referring page was
                        transferred with a secure protocol.*/
                        String referer =
                            requestHeaders.getPropertyValue("referer");
                        boolean sendReferrer = true;
                        if (referer != null && referer.startsWith("https:") &&
                            !absoluteUrl.startsWith("https:")) {
                            sendReferrer = false;
```

```
                }
                int size = requestHeaders.size();
                for (int i = 0; i < size; i++) {
                   String header = requestHeaders.getPropertyKey(i);
                   // remove refer header if needed
                   if ( !sendReferrer && header.equals("referer")) {
                      requestHeaders.removeProperty(i);
                      continue;
                   }
                   conn.setRequestProperty( header,
                      requestHeaders.getPropertyValue( i ) );
                }
             }
          } catch (IOException e1) {
          } finally {
             if (out != null) {
                try {
                   out.close();
                } catch (IOException e2) {
                }
             }
          }
          BrowserField browserField = getBrowserField(conn, e);
          _callbackApplication.displayBrowserField(browserField);
       break;
    }
    case Event.EVENT_BROWSER_FIELD_CHANGED : {
       /*
        * Browser field title might have changed. Update title.
        */
       break;
    }
    case Event.EVENT_REDIRECT : {
       RedirectEvent e = (RedirectEvent) event;
       switch (e.getType()) {
          case RedirectEvent.TYPE_SINGLE_FRAME_REDIRECT :
          // show redirect message
          Application.getApplication().invokeAndWait(new Runnable() {
             public void run() {
                Status.show("");
             }
          });
          break;
          case RedirectEvent.TYPE_JAVASCRIPT :
          case RedirectEvent.TYPE_META :
          case RedirectEvent.TYPE_300_REDIRECT :
       }
       String absoluteUrl = e.getLocation();
       HttpConnection conn = null;
       try {
          conn = (HttpConnection) Connector.open(absoluteUrl);
       } catch (IOException e1) {
       }
       BrowserField browserField = getBrowserField(conn,
          e.getOriginalEvent());
```

```
            _callbackApplication.displayBrowserField(browserField);
            break;
        }
        case Event.EVENT_CLOSE :
        // close the appication
        break;
        case Event.EVENT_TICK_CONTENT_READ : // no progress bar is supported
        case Event.EVENT_SET_HEADER :// no cache support
        case Event.EVENT_SET_HTTP_COOKIE :   // no cookie support
        case Event.EVENT_HISTORY :   // no history support
        case Event.EVENT_LOADING_IMAGES :// no progress bar is supported
        case Event.EVENT_EXECUTING_SCRIPT :  // no progress bar is supported
        case Event.EVENT_FULL_WINDOW :   // no full window support
        case Event.EVENT_STOP :  // no stop loading support
        default :
    }
    return null;
}
```

Retrieve browser content for rendering

Implement `getBrowserField()` so that it retrieves the browser content for rendering. The `getBrowserField()` method invokes `RenderingSession.getBrowserField()` to retrieve a browser field; applications cannot instantiate `BrowserField` directly.

```
public BrowserField getBrowserField (String absoluteUrl) {
    HttpConnection conn = null;
    try {
        conn = (HttpConnection) Connector.open(absoluteUrl);
        // set transcode to true
        conn.setRequestProperty( "x-rim-transcode-content", "*/*" );
    } catch (IOException e1) {
    }
    return getBrowserField(conn, null);
}
private BrowserField getBrowserField(HttpConnection conn, Event e) {
    BrowserField field = null;
    try {
        field = _renderingSession.getBrowserField(conn, this, e);
    } catch (RenderingException re) {
        return null;
    }
    // add to the event queue a thread that invokes finishLoading()
    Application.getApplication().invokeLater(new RenderingThread(field));
    return field;
}
```

Render the browser field

Invoke `BrowserField.finishLoading()`. In the following code sample, `BrowserField.getBrowserField()` runs `finishLoading()` by adding a new rendering thread to the application event queue.

Notes: HTML files display a blank field until you invoke `BrowserField.finishLoading()`. WML files and images might load before this method is invoked.

The `finishLoading()` method should be run on a separate thread so that the UI does not lock.

```
class RenderingThread implements Runnable {
    BrowserField _browserField;
    RenderingThread(BrowserField field) {
        _browserField = field;
    }
    public void run() {
        try {
            _browserField.finishLoading();
        } catch (RenderingException e) {
            // handle exception
        }

    }
}
```

Display the browser field

Implement `displayBrowserField()` so it deletes the screen contents, and then adds the browser field to the screen.

```
public void displayBrowserField(BrowserField browserField) {
    synchronized (Application.getEventLock()) {
        _vfm.deleteAll();
        _vfm.add(browserField);
    }
}
```

Code example

The following example consists of three files: BasicRenderingApplication.java, BrowserFieldHandlerApplication.java, and BrowserFieldSampleApp.java.

Example: Browser field sample

```
/**
 * BasicRenderingApplication.java
 * Copyright (C) 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.browser;

import java.io.IOException;
import java.io.OutputStream;
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import net.rim.device.api.browser.field.*;
import net.rim.device.api.io.http.HttpHeaders;
import net.rim.device.api.io.http.HttpProtocolConstants;
import net.rim.device.api.system.Application;
import net.rim.device.api.ui.Graphics;
import net.rim.device.api.ui.component.Status;
import com.rim.samples.docs.browser.BrowserFieldHandlerApplication.*;

final public class BasicRenderingApplication implements RenderingApplication {
    RenderingSession _renderingSession;
```

```
BrowserFieldHandlerApplication _callbackApplication;
public BasicRenderingApplication(BrowserFieldHandlerApplication
callBackApplication) {
    _renderingSession = RenderingSession.getNewInstance();
    _callbackApplication = callBackApplication;
}
/**
* Simple method to get a browser field by specifying the URL.
* This call blocks until the browser field is returned by
RenderingSession.getBrowserField().
* The browser field can continue to be rendered after the field is returned.
*
* @param absoluteUrl - absolute url of the page to render
* @return rendered browser field
*/
public BrowserContent getBrowserField (String absoluteUrl) {
    HttpConnection conn = null;
    try {
        conn = (HttpConnection) Connector.open(absoluteUrl);
        // Set transcode to true.
        conn.setRequestProperty( "x-rim-transcode-content", "*/*" );
    } catch (IOException e1) {
    }
    return getBrowserField(conn, null);
}
private BrowserContent getBrowserField(HttpConnection conn, Event e) {
    BrowserContent field = null;
    try {
        field = _renderingSession.getBrowserContent(conn, this, e);
    } catch (RenderingException re) {
        return null;
    }
    Application.getApplication().invokeLater(new RenderingThread(field));
    return field;
}

/**
* Invoked when an event occurs.
*/
public Object eventOccurred(Event event) {
    int eventId = event.getUID();
    switch (eventId) {
    case Event.EVENT_URL_REQUESTED : {
        UrlRequestedEvent e = (UrlRequestedEvent) event;
        // this is a regular request
        String absoluteUrl = e.getURL();
        HttpConnection conn = null;
        OutputStream out = null;
        try {
            conn = (HttpConnection) Connector.open(absoluteUrl);
                byte[] postData = e.getPostData();
                if (postData == null) {
                    conn.setRequestMethod(HttpConnection.GET);
                } else {
                    conn.setRequestMethod(HttpConnection.POST);
```

```
                    conn.setRequestProperty(
HttpProtocolConstants.HEADER_CONTENT_LENGTH,
                        String.valueOf(postData.length));
                    out = conn.openOutputStream();
                    out.write(postData);
                }
                HttpHeaders requestHeaders = e.getHeaders();
                if (requestHeaders != null) {
                    /* From
                    http://www.w3.org/Protocols/rfc2616/rfc2616-
sec15.html#sec15.1.3
                    ...
                    Clients SHOULD NOT include a Referer header field in a
                    (non-secure) HTTP request if the referring page was
                    transferred with a secure protocol.*/
                  String referer = requestHeaders.getPropertyValue("referer");
                    boolean sendReferrer = true;
                    if (referer != null && referer.startsWith("https:") &&
!absoluteUrl.startsWith("https:")) {
                        sendReferrer = false;
                    }
                    int size = requestHeaders.size();
                    for (int i = 0; i < size; i++) {
                        String header = requestHeaders.getPropertyKey(i);
                        // Remove refer header if needed.
                        if ( !sendReferrer && header.equals("referrer")) {
                            requestHeaders.removeProperty(i);
                            continue;
                        }
                        conn.setRequestProperty( header,
requestHeaders.getPropertyValue( i ) );
                    }
                }
            } catch (IOException e1) {
            } finally {
                if (out != null) {
                    try {
                        out.close();
                    } catch (IOException e2) {
                    }
                }
            }
            BrowserContent browserField = getBrowserField(conn, e);
            _callbackApplication.displayBrowserField(browserField);
        break;
    }
    case Event.EVENT_BROWSER_CONTENT_CHANGED : {
        // Browser field title might have changed. Update title.
        break;
    }
    case Event.EVENT_REDIRECT : {
        RedirectEvent e = (RedirectEvent) event;
        switch (e.getType()) {
            case RedirectEvent.TYPE_SINGLE_FRAME_REDIRECT :
            // Show redirect message.
            Application.getApplication().invokeAndWait(new Runnable() {
```

```
                public void run() {
                    Status.show("");
                }
            });
            break;
            case RedirectEvent.TYPE_JAVASCRIPT :
            case RedirectEvent.TYPE_META :
            case RedirectEvent.TYPE_300_REDIRECT :
        }
        String absoluteUrl = e.getLocation();
        HttpConnection conn = null;
        try {
            conn = (HttpConnection) Connector.open(absoluteUrl);
        } catch (IOException e1) {
        }
        BrowserContent browserField = getBrowserField(conn,
e.getOriginalEvent());
        _callbackApplication.displayBrowserField(browserField);
        break;
    }
    case Event.EVENT_CLOSE :
    // Close the appication.
    break;

    case Event.EVENT_TICK_CONTENT_READ : // No progress bar is supported.
    case Event.EVENT_SET_HEADER :        // No cache support.
    case Event.EVENT_SET_HTTP_COOKIE :   // No cookie support.
    case Event.EVENT_HISTORY :           // No history support.
    case Event.EVENT_LOADING_IMAGES :    // No progress bar is supported.
    case Event.EVENT_EXECUTING_SCRIPT :  // No progress bar is supported.
    case Event.EVENT_FULL_WINDOW :       // No full window support.
    case Event.EVENT_STOP :              // No stop loading support.
    default :
    }
    return null;
}
/**
 * Retrieves the pixels of height available for provided browser content.
 * @param browserField Content for which to retrieve available pixels of
height.
 * @return Height available.
 */
public int getAvailableHeight(BrowserContent browserField) {
    // Field has full screen.
    return Graphics.getScreenHeight();
}
/**
* Retrieves the pixels of width available for provided browser content.
*/
public int getAvailableWidth(BrowserContent browserField) {
    // Field has full screen.
    return Graphics.getScreenWidth();
}
/**
* Retrieves the history position for provided browser content.
*/
```

```
    public int getHistoryPosition(BrowserContent browserField) {
        // No history support.
        return 0;
    }
    /**
    *  Retrieves cookies associated with a provided URL.
    */
    public String getHTTPCookie(String url) {
        // No cookie support.
        return null;
    }
    /**
    * Retrieves the specified resource.
    */
    public HttpConnection getResource( RequestedResource resource, BrowserContent
   referrer) {
        if (resource == null) {
            return null;
        }
        // Check if this is cache-only request.
        if (resource.isCacheOnly()) {
            // No cache support.
            return null;
        }
        String url = resource.getUrl();
        if (url == null) {
            return null;
        }
        // If referrer is null, return the connection.
        if (referrer == null) {
            HttpConnection conn;
        try {
            return (HttpConnection) Connector.open(resource.getUrl());
        } catch (IOException e) {
            return null;
        }
        } else {
        // If referrer is provided, set up the connection on a
        // separate thread.
        Application.getApplication().invokeLater(
            new RetrieveThread(resource, referrer));
        }
        return null;
    }

    /**
    * Invokes the provided runnable object.
    */
    public void invokeRunnable(Runnable runnable) {
        (new Thread(runnable)).run();
    }
}
class RenderingThread implements Runnable {
    BrowserContent _browserField;
    RenderingThread(BrowserContent field) {
        _browserField = field;
```

```
    }
    public void run() {
        try {
            _browserField.finishLoading();
        } catch (RenderingException e) {
        }
    }
}

class RetrieveThread implements Runnable {
    BrowserContent _browserField;
    RequestedResource _resource;
    RetrieveThread(RequestedResource resource, BrowserContent referrer) {
        _browserField = referrer;
        _resource = resource;
    }
    public void run() {
        HttpConnection conn;
        try {
            conn = (HttpConnection) Connector.open(_resource.getUrl());
        } catch (IOException e) {
            return;
        }
        _resource.setHttpConnection(conn);
        _browserField.resourceReady(_resource);
    }
}
/**
 * BrowserFieldSampleApp.java
 * Copyright (C) 2002-2004 Research In Motion Limited.
 */

package com.rim.samples.docs.browser;

import net.rim.device.api.browser.field.BrowserContent;
import net.rim.device.api.system.Application;
import net.rim.device.api.ui.Manager;
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.container.VerticalFieldManager;

import com.rim.samples.docs.browser.BasicRenderingApplication;
import com.rim.samples.docs.browser.BrowserFieldHandlerApplication;

public final class BrowserFieldSampleApp extends UiApplication implements
   BrowserFieldHandlerApplication {

    VerticalFieldManager _vfm;

    // Constructor.
    BrowserFieldSampleApp() {
        MainScreen mainScreen = new MainScreen();
        _vfm = new VerticalFieldManager(Manager.VERTICAL_SCROLL |
   Manager.VERTICAL_SCROLLBAR);
        mainScreen.add(_vfm);
        pushScreen(mainScreen);
```

```
        // get and display the browser field on a separate thread
        this.invokeLater((new CreationThread(this)));
    }

    // Callback method for basic rendering application. Displays rendered browser
   field.
    // @param browserField - browser field to display
    public void displayBrowserField(BrowserContent browserField) {
        synchronized (Application.getEventLock()) {
            _vfm.deleteAll();
            _vfm.add(browserField.getDisplayableContent());
        }
    }

    public static void main(String[] args) {
        BrowserFieldSampleApp app = new BrowserFieldSampleApp();
        app.enterEventDispatcher();
    }
}
class CreationThread extends Thread {
    BrowserFieldHandlerApplication _callBackApplication;
    BasicRenderingApplication      _renderingApplication;
    public CreationThread(BrowserFieldHandlerApplication callBackApplication) {
        _callBackApplication = callBackApplication;
    }
    public void run() {
        _renderingApplication = new
   BasicRenderingApplication(_callBackApplication);
        BrowserContent field =
   _renderingApplication.getBrowserField("www.rim.com");
        _callBackApplication.displayBrowserField(field);
    }
}
```

Supporting additional MIME types

The browser page API, in the `net.rim.device.api.browser.plugin` package, enables third-party applications to register themselves with the rendering library as rendering providers for specific MIME types that are not currently supported by the BlackBerry Browser.

Note: The `BrowserFieldProviderRegistry.register()` method throws an exception if you try to register a MIME type that is already supported by the BlackBerry Browser. For a list of supported MIME types, invoke `RenderingSession.getSupportedMimeType()`.

Register as rendering provider for a MIME type

To support additional MIME types, extend the `BrowserContentProvider` abstract class. To specify display characteristics such as no scroll bar or full screen display, implement the `BrowserPageContext` interface.

The rendering library invokes `BrowserContentProvider.getAccept()` and `BrowserContentProvider.getSupportedMimeTypes()` to identify the MIME types the provider renders.

List accepted MIME types

Implement `getAccept()` and `getSupportedMimeTypes()` to list the list of MIME types the provider can accept given a set of rendering options. The `getAccept()` method considers the rendering options that have been set; this example assumes that no rendering options are set. See `RenderingOptions` in the *API Reference* for more information.

```
public String[] getAccept(RenderingOptions context) {
   // Return subset of getSupportedMimeTypes() if accept depends in rendering
   // options. For example HTML can be disabled in the rendering options, and
   // HTMLConverter would remove html mime types.
   return ACCEPT;
}
public String[] getSupportedMimeTypes() {
   return ACCEPT;
}
```

Specify display characteristics

Implement the `BrowserPageContext` interface. If you do not implement this interface, default values are used.

In this example, the properties are all integers. An application with `Boolean`, `String`, and `Object` properties would implement the corresponding methods.

```
public boolean getPropertyWithBooleanValue(int id, boolean defaultValue) {
   // TODO Auto-generated method stub
   return false;
}
public int getPropertyWithIntValue(int id, int defaultValue) {
   if (id == BrowserPageContext.DISPLAY_STYLE) {
      // Disable the scroll bar.
      return BrowserPageContext.STYLE_NO_VERTICAL_SCROLLBAR;
   }
   return 0;
}
public Object getPropertyWithObjectValue(int id, Object defaultValue) {
   // TODO Auto-generated method stub
   return null;
}
public String getPropertyWithStringValue(int id, String defaultValue) {
   // TODO Auto-generated method stub
   return null;
}
```

Retrieve a field to render the content

```
public BrowserContent getBrowserContent( BrowserContentProviderContext context)
   throws RenderingException {

   if (context == null) {
      throw new RenderingException("No Context is passed into Provider");
```

```
    }

    BrowserContentBaseImpl browserContentBaseImpl = new
        BrowserContentBaseImpl(context.getHttpConnection().getURL(), null,
        context.getRenderingApplication(),
        context.getRenderingSession().getRenderingOptions(), context.getFlags());
    VerticalFieldManager vfm = new VerticalFieldManager(Manager.VERTICAL_SCROLL);
    vfm.add(new LabelField("Mime type: ") );
    vfm.add(new LabelField(ACCEPT[0]));
    vfm.add(new SeparatorField());
    vfm.add(new LabelField("Content of the resource file: \n"));
    vfm.add(new SeparatorField());
    try {
        HttpConnection conn = context.getHttpConnection();
        InputStream in = conn.openInputStream();
        byte[] data = IOUtilities.streamToBytes(in);
        vfm.add(new LabelField(new String(data)));

    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    browserContentBaseImpl.setContent(vfm);
    browserContentBaseImpl.setTitle(ACCEPT[0]);
    // Set browser page context, this will tell the browser how to display this
    // field.
    browserContentBaseImpl.setBrowserPageContext(this);
    return browserContentBaseImpl;
}
```

Register when the handheld starts

Create a library project and set its properties to auto-run on startup. In `libMain()`, invoke `BrowserFieldProviderRegistry.getInstance()` and then invoke `register()`.

```
public static void libMain( String[] args ) {
    BrowserContentProviderRegistry converterRegistry =
    BrowserContentProviderRegistry.getInstance();
    if (converterRegistry != null) {
        converterRegistry.register(new BrowserPlugin());
    }
}
```

Code example

Example: BrowserPlugin.java and LoaderApp.java

```
/**
 * LoaderApp.java
 * Copyright (C) 2004 Research In Motion Limited.
 */
package com.rim.samples.docs.browser;
import net.rim.device.api.browser.plugin.BrowserContentProviderRegistry;
```

```
final class LoaderApp {

    public static void libMain( String[] args ) {
        BrowserContentProviderRegistry converterRegistry =
   BrowserContentProviderRegistry.getInstance();
        if (converterRegistry != null) {
            converterRegistry.register(new BrowserPlugin());
        }
    }
}
/**
 * BrowserPlugin.java
 * Copyright (C) 2004 Research In Motion Limited.
 */
package com.rim.samples.docs.browser;

import java.io.IOException;
import java.io.InputStream;

import javax.microedition.io.HttpConnection;

import net.rim.device.api.browser.field.*;
import net.rim.device.api.browser.plugin.*;
import net.rim.device.api.ui.Manager;
import net.rim.device.api.ui.component.LabelField;
import net.rim.device.api.ui.component.SeparatorField;
import net.rim.device.api.ui.container.VerticalFieldManager;

/**
 * Create a file with xxtest extension and associate that type with
 * application/x-vnd.rim.xxxtest mime type on any server.
 */
public final class BrowserPlugin  extends BrowserContentProvider implements
   BrowserPageContext {

    private static final String[] ACCEPT = {"application/x-vnd.rim.xxxtest"};

    /**
   * Retrieves list of mime types this provider can accept given a set of rendering
   options.
    * @param context Rendering options in place this provider should consider.
    * @return Array of mime types this provider will accept, given the provided
   rendering options.
     */
    public String[] getAccept(RenderingOptions context) {
       // Return subset of getSupportedMimeTypes() if accept depends in rendering
   options.
        // For example HTML can be disabled in the rendering options, and
   HTMLConverter would remove
        // html MIME types.
        return ACCEPT;
    }
    /**
     * Retrieves a browser content capable of rendering the mime content this
   provider can handle.
     * @param context Provider context object provided by rendering session.
```

```
     * @return Browser content to render specialized content.
     */

    public BrowserContent getBrowserContent( BrowserContentProviderContext context)
    throws RenderingException {

        if (context == null) {
            throw new RenderingException("No Context is passed into Provider");
        }

        BrowserContentBaseImpl browserContentBaseImpl = new
    BrowserContentBaseImpl(context.getHttpConnection().getURL(),
            null, context.getRenderingApplication(),
    context.getRenderingSession().getRenderingOptions(), context.getFlags());

        VerticalFieldManager vfm = new
    VerticalFieldManager(Manager.VERTICAL_SCROLL);


        vfm.add(new LabelField("Mime type: ") );
        vfm.add(new LabelField(ACCEPT[0]));
        vfm.add(new SeparatorField());
        vfm.add(new LabelField("Content of the resource file: \n"));
        vfm.add(new SeparatorField());

        try {
            HttpConnection conn = context.getHttpConnection();
            InputStream in = conn.openInputStream();
            int numBytes = in.available();
            byte[] data = new byte[numBytes];
            in.read(data, 0, numBytes);
            vfm.add(new LabelField(new String(data)));

        } catch (IOException e) {
            e.printStackTrace();
        }
        browserContentBaseImpl.setContent(vfm);
        browserContentBaseImpl.setTitle(ACCEPT[0]);
        // Set browser page context. This tells the browser how to display this
    field.
        browserContentBaseImpl.setBrowserPageContext(this);

        return browserContentBaseImpl;
    }
    /**
     * Retrieves all the mime content types supported by this provider.
     * @return Mime types this converter supports.
     */

    public String[] getSupportedMimeTypes() {
        return ACCEPT;
    }


    /**
     * Retrieves value of specified property as a boolean value.
```

```
     * @param id ID of property to query.
     * @param defaultValue Expected default value of property.
     * @return Current value of property.
     */

    public boolean getPropertyWithBooleanValue(int id, boolean defaultValue) {
        return false;
    }
    /**
     * Retrieves value of specified property as an int.
     * @param id ID of property to query.
     * @param defaultValue Expected default value of property.
     * @return Current value of property.
     */
    public int getPropertyWithIntValue(int id, int defaultValue) {

        if (id == BrowserPageContext.DISPLAY_STYLE) {
            // Disable the scroll bar.
            return BrowserPageContext.STYLE_NO_VERTICAL_SCROLLBAR;
        }

        return 0;
    }
    /**
     * Retrieves value of specified property as an object.
     * @param id ID of property to query.
     * @param defaultValue Expected default value of property.
     * @return Current value of property.
     */

    public Object getPropertyWithObjectValue(int id, Object defaultValue) {
        return null;
    }
    /* Retrieves value of specified property as a String value.
     * @param id - ID of property to query.
     * @param defaultValue - Expected default value of property.
     * @return Current value of property.
     */
    public String getPropertyWithStringValue(int id, String defaultValue) {
        return null;
    }
}
```

Registering as a HTTP filter

The HTTP filter API (`net.rim.device.api.io.http`) enables an application to register with the browser as provider for a specific URL. When users type in the specified URL, the connection stack is rerouted to the specified application.

Note: Check for a `ControlledAccessException` when your application first accesses the HTTP filter API. This runtime exception is thrown if the system administrator restricts access to the HTTP filter API using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

Register as an HTTP filter

Invoke `HttpFilterRegistry.registerFilter()`. Provide as parameters the URL to intercept and the package name of the application that defines interception behaviour.

```
HttpFilterRegistry.registerFilter("content.blackberry.com",
    "com.rim.samples.device.httpfilterdemo.precanned");
```

Perform registration at startup

Create a library project and set its properties to auto-run on startup. Invoke `registerFilter()` in `libMain()`.

Code example

The following example consists of two files, PackageManager.java, which registers the filter on startup, and Protocol.java, which defines the filter behavior.

Example: HTTP filter example

```
/**
 * PackageManager.java
 * Copyright (C) 2004 Research In Motion Limited. All rights reserved.
 */
package com.rim.samples.docs.httpfilterdemo;

import net.rim.device.api.io.http.HttpFilterRegistry;

/**
 * This class runs on startup of the device and registers the necessary http
   filters.
 */
final class PackageManager
{
    public static void libMain(String[] args) {
        HttpFilterRegistry.registerFilter("www.blackberry.com",
   "com.rim.samples.docs.httpfilterdemo.filter");
    }
}
/**
 * Protocol.java
 * Copyright (C) 2004 Research In Motion Limited. All rights reserved.
 */
package com.rim.samples.docs.httpfilterdemo.filter;
```

```
import net.rim.device.api.io.FilterBaseInterface;
import java.io.*;
import javax.microedition.io.*;

/**
 * This class implements a simple pass through mechanism that writes out the http
   response headers to System.out.
 */
public final class Protocol implements FilterBaseInterface, HttpConnection {
    private HttpConnection _subConnection;

    /**
     * Defined by FilterBaseInterface.
     * This method opens a filtered Http Connection.
     */
    public Connection openFilter( String name, int mode, boolean timeouts ) throws
   IOException {
        _subConnection = (HttpConnection)Connector.open("http:" + name +
   ";usefilter=false", mode, timeouts);
        if (_subConnection != null) {
            return this;
        }

        // Filed to open the sub connection; so let us fail too.
        return null;
    }

    // Return a string representation of the URL for this connection.
    public String getURL() {
        return _subConnection.getURL();
    }

    // Returns the protocol name of the URL of this HttpConnection. e.g., http or
   https.
    public String getProtocol() {
        return _subConnection.getProtocol();
    }

    // Returns the host information of the URL of this HttpConnection. e.g. host
   name or IPv4 address.
    public String getHost() {
        return _subConnection.getHost();
    }

    // Returns the file portion of the URL of this HttpConnection.
    public String getFile() {
        return _subConnection.getFile();
    }

    /**
     * Returns the ref portion of the URL of this HttpConnection.
     * RFC2396 specifies the optional fragment identifier as the the text after the
   crosshatch (#)
     * character in the URL. This information may be used by the user agent as
   additional
```

```
  * reference information after the resource is successfully retrieved. The
format and
  * interpretation of the fragment identifier is dependent on the media
type[RFC2046] of
  * the retrieved information.
  */
 public String getRef() {
     return _subConnection.getRef();
 }

 // Returns the network port number of the URL for this HttpConnection.
 public int getPort() {
     return _subConnection.getPort();
 }

 // Returns the query portion of the URL of this HttpConnection.
 // RFC2396 defines the query component as the text after the first
 // question-mark (?) character in the URL.
 public String getQuery() {
     return _subConnection.getQuery();
 }

 // Get the current request method. e.g. HEAD, GET, POST The default value is
GET.
 public String getRequestMethod() {
     return _subConnection.getRequestMethod();
 }

 // Set the method for the URL request, one of: GET, POST, HEAD, subject to
protocol restrictions. The default method is GET.
 public void setRequestMethod(String method) throws IOException {
     _subConnection.setRequestMethod(method);
 }

 // Returns the value of the named general request property for this connection.
 public String getRequestProperty(String key) {
     return _subConnection.getRequestProperty(key);
 }

 // Sets the general request property. If a property with the key already
exists, overwrite its value with the new value.
 // Note: HTTP requires all request properties which can legally have multiple
instances with the
 // same key to use a comma-separated list syntax which enables multiple
properties to be appended into a single property.
 public void setRequestProperty(String key, String value) throws IOException {
    System.out.println("Request property <key, value>: " + key + ", " + value );
     _subConnection.setRequestProperty(key, value);
 }

 // Returns the HTTP response status code.
 public int getResponseCode() throws IOException {
     return _subConnection.getResponseCode();
 }
```

```
 // Gets the HTTP response message, if any, returned along with the response
code from a server.
 public String getResponseMessage() throws IOException {
     return _subConnection.getResponseMessage();
 }

 // Returns the value of the expires header field.
 public long getExpiration() throws IOException {
     return _subConnection.getExpiration();
 }

 // Returns the value of the date header field.
 public long getDate() throws IOException {
     return _subConnection.getDate();
 }

 // Returns the value of the last-modified header field. The result is the
number of milliseconds since January 1, 1970 GMT.
 public long getLastModified() throws IOException {
     return _subConnection.getLastModified();
 }

 // Returns the value of the named header field.
 public String getHeaderField(String name) throws IOException
 {
     String value = _subConnection.getHeaderField(name);
    System.out.println("Response property <key, value>: " + name + ", " + value
);
     return value;
 }

 // Returns the value of the named field parsed as a number.
 public int getHeaderFieldInt(String name, int def) throws IOException {
     return _subConnection.getHeaderFieldInt(name, def);
 }

 // Returns the value of the named field parsed as date. The result is the
 // number of milliseconds since January 1, 1970 GMT represented by the named
field.
 public long getHeaderFieldDate(String name, long def) throws IOException {
     return _subConnection.getHeaderFieldDate(name, def);
 }


 // Gets a header field value by index.
 public String getHeaderField(int n) throws IOException {
     return _subConnection.getHeaderField(n);
 }

 // Gets a header field key by index.
 public String getHeaderFieldKey(int n) throws IOException {
     return _subConnection.getHeaderFieldKey(n);
 }

 // Returns the type of content that the resource connected to is providing.
 public String getType() {
```

```
            return _subConnection.getType();
        }

        // Returns a string describing the encoding of the content which the resource
        connected to is providing.
        public String getEncoding() {
            return _subConnection.getEncoding();
        }

        // Returns the length of the content which is being provided.
        public long getLength() {
            return _subConnection.getLength();
        }

        // Opens and returns an input stream for a connection.
        public InputStream openInputStream() throws IOException {
            return _subConnection.openInputStream();
        }

        // Opens and returns a data input stream for a connection.
        public DataInputStream openDataInputStream() throws IOException {
            return _subConnection.openDataInputStream();
        }

        // Opens and returns an output stream for a connection.
        public OutputStream openOutputStream() throws IOException {
            return _subConnection.openOutputStream();
        }

        // Opens and returns a data output stream for a connection.
        public DataOutputStream openDataOutputStream() throws IOException {
            return _subConnection.openDataOutputStream();
        }

        // Closes the connection.
        public void close() throws IOException {
            _subConnection.close();
        }
    }
```

6

Accessing the phone application

- Using the phone API
- Listening for phone events
- Accessing and managing phone logs

Using the phone API

The phone API (`net.rim.blackberry.api.phone`) provides access to advanced features of the phone application, such as enabling applications to inject DTMF tones into active calls.

Tip: To simply invoke the phone application and place a phone call, use the invocation API (`net.rim.blackberry.api.invoke`). See " Starting BlackBerry applications" on page 75 for more information.

Note: Check for a `ControlledAccessException` when your application first accesses the phone API. This runtime exception is thrown if the system administrator restricts access to the phone API using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

Retrieve a phone call

To retrieve the active phone call, invoke `Phone.getActiveCall()`. To retrieve a phone call by call ID, invoke `Phone.getCall()`.

```
PhoneCall call = Phone.getActiveCall();
```

Retrieve phone call information

The `PhoneCall` class provides methods that enable applications to retrieve information about phone calls. For example, the following code sample checks the length of the call, the status of the call, and whether it is outgoing before displaying a message that includes the display phone number.

```
int threshold = 120; // Alert user if outgoing calls last longer than threshold.
int elapsedTime = call.getElapsedTime();
// Use getStatusString() to retrieve status as an string.
int status = call.getStatus();
if ((status == PhoneCall.STATUS_CONNECTED || status ==
   PhoneCall.STATUS_CONNECTING) && call.isOutGoing() && elapsedTime > threshold) {
   // Use getCallId() to retrieve the caller ID as as an integer.
   String phoneNumber = call.getDisplayPhoneNumber();
   Status.show("Your call to " + phoneNumber + " has lasted more than " +
      (String)threshold + ".");
}
```

Add DTMF tones

To add a single Dual Tone Multi Frequency (DTMF, or touch-tone) tone to the send queue, invoke `sendDTMFTone()`. To add multiple tones to the send queue, invoke `sendDTMFTones()`.

Note: Handhelds play DTMF tones as soon as no other tones are pending, overriding conversations.

Retrieve the send queue for the current call

Invoke `getDTMFTones()`.

Listening for phone events

Implement the `PhoneListener` interface and then invoke `Phone.addPhoneListener()` to register the phone listener with the system.

To de-register a phone listener, invoke `removePhoneListener()`.

To act on a particular event, implement one of the following methods.

Method	Description
`callAdded(int callId)`	This method is invoked when a call is added to a conference call.
`callAnswered(int callId)`	This method is invoked when a user answers a call (user driven).
`callConferenceCallEstablished(int callId)`	This method is invoked when a conference call is established.
`callConnected(int callId)`	This method is invoked when the network indicates a connected event (network driven).
`callDirectConnectConnected(int callId)`	This method is invoked when a direct-connect call is connected.
`callDirectConnectDisconnected(int callId)`	This method is invoked when a direct-connect call is disconnected.
`callDisconnected(int callId)`	This method is invoked when a call is disconnected.
`callEndedByUser(int callId)`	This method is invoked when a user ends the call.
`callFailed(int callId, int reason)`	This method is invoked when a call fails.
`callHeld(int callId)`	This method is invoked when a call goes on hold.
`callIncoming(int callId)`	This method is invoked when a new call arrives.
`callInitiated(int callid)`	This method is invoked when an outgoing call is initiated by the handheld.
`callRemoved(int callId)`	This method is invoked when a call is removed from a conference call.
`callResumed(int callId)`	This method is invoked when a held call resumes.
`callWaiting(int callid)`	This method is invoked when a call is waiting.
`conferenceCallDisconnected(int callId)`	This method is invoked when a conference call is ended (all members are disconnected).

Accessing and managing phone logs

The phone logs API (`net.rim.blackberry.api.phone.phonelogs`) enables applications to access the phone application log files. The phone call history consists of call logs, which represent individual phone calls, grouped into a phone log.

Retrieve phone logs

The `PhoneLogs` class represents the phone call history. It provides method that enables you to open, add, delete, or swap call logs.

To retrieve a phone log, invoke `PhoneLogs.getInstance()`.

```
PhoneLogs _logs = PhoneLogs.getInstance();
```

Retrieve the number of calls in a folder

The phone logs are divided into two folders: `FOLDER_NORMAL_CALLS` and `FOLDER_MISSED_CALLS`. To retrieve the number of calls in a folder, invoke `numberOfCalls()`.

```
int numberOfCalls = _logs.numberOfCalls(FOLDER_NORMAL_CALLS);
```

Retrieve a call log

Invoke `PhoneLogs.callAt()`.

You can instantiate two types of call logs: `PhoneCallLog` objects, which can only have one participant, and `ConferencePhoneCallLog` objects, which have two or more participants. These objects enable you to retrieve or change call log information, such as the participants or the date of the call.

```
PhoneCallLog phoneLog = (PhoneCallLog)_logs.callAt(0);
```

Retrieve a call participant

The `PhoneCallLogID` class identifies participants in phone call log by phone number. Invoke `PhoneCallLog.getParticipant()` or `ConferencePhoneCallLog.getParticipantAt()` to retrieve a participant.

```
PhoneCallLogID participant = phoneLogs.getParticipant();
```

Add a call log

Tip: The `PhoneCallLogID` constructor strips dashes and other non-numeric characters from phone numbers.

To create a new phone or conference call log, invoke the `PhoneCallLog()` or `ConferencePhoneCallLog()` constructor. Provide as parameters the date, duration, participants and notes for the call.

```
Date date = new Date("1000"); // date of call
int duration = 60; // duration of call
PhoneCallLogID caller1 = new PhoneCallLogID("555-1234"); // first participant
PhoneCallLogID caller2 = new PhoneCallLogID("555-1235"); // second participant
String notes = "New call."; // notes
ConferencePhoneCallLog conferenceCall = new ConferencePhoneCallLog(date, duration,
   PhoneLogs.FOLDER_NORMAL_CALLS, caller1, caller2, notes);
```

To add this object to the phone log at the next available index, invoke `PhoneLogs.addCall()`.

```
_logs.addCall(conferenceCall);
```

To replace the call log at a given index with a new call log, invoke `PhoneLogs.swapCall()`.

```
_logs.swapCall(conferenceCall, 0, FOLDER_NORMAL_CALLS);
```

Note: The `swapCall()` method deletes the call at the given index.

Delete a call log

Invoke `PhoneLogs.deleteCall()`.

```
_logs.deleteCall(0);
```

Code example

The following code example calculates the time spent on the phone with a given participant.

Example: PhoneLogsDemo.java

```
/**
 * PhoneLogsDemo.java
 * Copyright (C) 2001-2004 Research In Motion Limited. All rights reserved.
 */
package com.rim.samples.docs.phonelogs;

import net.rim.blackberry.api.phone.phonelogs.*;
import java.lang.*;

public class PhoneLogsDemo {
    private PhoneLogs _logs;
    static public void main(String[] args) {
        PhoneLogsDemo phoneLogsDemo = new PhoneLogsDemo();
        PhoneCallLogID participant = new PhoneCallLogID("5551234");
        int timeSpokenTo = phoneLogsDemo.findTimeSpokenTo(participant,
            PhoneLogs.FOLDER_NORMAL_CALLS);
    }
    private PhoneLogsDemo() {
        _logs = PhoneLogs.getInstance();
    }
    public void close() {
    }
    // Returns the number of seconds spent on the phone with a participant.
    public int findTimeSpokenTo(PhoneCallLogID participant,
        long folder) {
        int numberOfCalls = this._logs.numberOfCalls(folder);
        int timeSpokenTo = 0;
        PhoneCallLog phoneCallLog;
        ConferencePhoneCallLog conferencePhoneCallLog;
        for (int i = 0; i < numberOfCalls; i++) {
            Object o = _logs.callAt(i, folder);
            if (o instanceof PhoneCallLog) {
```

```
                phoneCallLog = (PhoneCallLog)_logs.callAt(i, folder);
                if ( phoneCallLog.getParticipant() == participant)
                    timeSpokenTo += phoneCallLog.getDuration();
            } else {
                conferencePhoneCallLog = (ConferencePhoneCallLog)_logs.callAt(i,
    folder);
                int participants = conferencePhoneCallLog.numberOfParticipants();
                for (int j = 0; j < participants; j++)
                  if (conferencePhoneCallLog.getParticipantAt(j) == participant) {
                       timeSpokenTo += conferencePhoneCallLog.getDuration();
                       j = participants;
                    }
            }
        }
        return timeSpokenTo;
    }
}
```

7

Communicating with BlackBerry applications

- Starting BlackBerry applications
- Adding menu items to BlackBerry applications
- Code example

Starting BlackBerry applications

The invocation APIs (`net.rim.blackberry.api.invoke`) enable applications to start standard BlackBerry applications.

Note: Check for a `ControlledAccessException` if your application invokes the phone. This runtime exception is thrown if the system administrator restricts access to the phone application using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

To start an application, call `Invoke.invokeApplication()` with the appropriate constant and an object of the appropriate `ApplicationArguments` subclass.

Note: Calling `Invoke.invokeApplication()` results in a process context switch. When the BlackBerry application is started, your application loses control. When the started application session ends, context might not return to your application.

Application	Constant	Class
Address book	APP_TYPE_ADDRESSBOOK	`AddressBookArguments`
Calendar	APP_TYPE_CALENDAR	`CalendarArguments`
Memo pad	APP_TYPE_MEMOPAD	`MemoArguments`
Messages	APP_TYPE_MESSAGES	`MessageArguments`
Phone	APP_TYPE_PHONE	`PhoneArguments`
Tasks	APP_TYPE_TASKS	`TaskArguments`

Tips: The BlackBerry Browser is invoked from the browser application API (`net.rim.blackberry.api.browser`). See " Display web content in the browser" on page 47 for more information.

The phone API (`net.rim.blackberry.api.phone`) provides access to advanced features of the phone application. See " Using the phone API" on page 65 for more information.

The following excerpt from the Restaurants.java code example creates a menu item that invokes the phone application to call a restaurant.

```
private MenuItem phoneItem = new MenuItem(_resources.getString(MENUITEM_PHONE),
   110, 12) {
   public void run() {
      synchronized(store) {
         String phoneNumber = phonefield.getText();
         if ( phoneNumber.length == 0 ) {
            Dialog.alert(_resources.getString(ALERT_NO_PHONENUMBER));
         } else {
            PhoneArguments call = new PhoneArguments(PhoneArguments.ARG_CALL,
               phoneNumber);
```

```
                Invoke.invokeApplication(Invoke.APP_TYPE_PHONE, call);
            }
        }
    }
};
```

Adding menu items to BlackBerry applications

The application menu item API, in the `net.rim.blackberry.api.menuitem` package, enables you to add menu items to BlackBerry applications.

For example, to integrate a customer relationship management application with the BlackBerry address book application, add a **View Sales Order** menu item. When users click the **View Sales Order** menu item, the application opens with a list of sales orders that the contact has placed.

The `ApplicationMenuItemRepository` class enables you to add or remove application menu items. It provides constants that define the application contexts in which a menu item can appear. For example, the `ApplicationMenuItemRepository.MENUITEM_MESSAGE_LIST` constant specifies that the menu item appears when the messages screen is open.

The abstract `ApplicationMenuItem` class defines an item that appears in an application menu. To create a menu item, extend the `ApplicationMenuItem` class.

Create an application menu item

To create an application menu item, extend the abstract `ApplicationMenuItem` class.

```
public class SampleMenuItem extends ApplicationMenuItem { ... }
```

Specify the position of the menu item in the menu

You can optionally override the constructor. In the following code sample, the constructor invokes `ApplicationMenuItem()` with an integer that specifies the relative order of the item in the menu (a higher number means that the menu item appears lower in the menu).

```
SampleMenuItem() {
    super(20);
}
```

Specify the menu item text

To specify the text that appears on the menu, implement `toString()`.

```
public String toString() {
    return "Open the Contacts Demo application";
}
```

Specify menu item behavior

To specify the behavior of the menu item, implement `run()`.

```
public Object run(Object context) {
    Contact c = (Contact)context; //an error if this doesn't work
```

```
    if ( c ! null ) {
        new ContactsDemo().enterEventDispatcher();
    } else {
        throw new IllegalStateException( "Context is null, expected a Contact
            instance");
    }
    Dialog.alert("Viewing an email message in the email view");
    return null;
}
```

Register the application menu item

Retrieve the application menu item repository

Invoke `ApplicationMenuItemRepository.getInstance()`.

```
ApplicationMenuItemRepository repository =
    ApplicationMenuItemRepository.getInstance();
```

Define a unique ID

Use the hash of the package name as the unique long ID for the application menu item repository.

```
long ID = 0x7cab1e23b72a0033L; //hash of com.rim.samples.docs.menuitem
```

Create your application menu item

Invoke the constructor.

```
TestApplicationMenuItem tami = new TestApplicationMenuItem();
```

Add the menu item to the repository

Invoke `addMenuItem()`.

```
repository.addMenuItem(ApplicationMenuItemRepository.MENUITEM_ADDRESSCARD_VIEW,
    tami);
```

Code example

The following code example creates a menu item that appears when a user views a contact in the address book. When a user clicks the menu item, the Contacts Demo application appears.

Example: DemoAppMenuItem.java

```
/**
 * DemoApplicationMenuItem.java
 * Copyright (C) 2003 Research In Motion Limited.
*/

package com.rim.samples.docs.menuitem;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.component.Dialog.*;
import net.rim.blackberry.api.menuitem.*;
import javax.microedition.pim.*;
```

```
import com.rim.samples.docs.contactsdemo.*;


public final class DemoAppMenuItem extends Application {
    private static long ID = 0x7cab1e23b72a0033L;
    //com.rim.samples.docs.menuitem

    public static void main(String[] args) {
        DemoAppMenuItem app = new DemoAppMenuItem();
        app.enterEventDispatcher();
    }

    DemoAppMenuItem() {
        ApplicationMenuItemRepository amir =
        ApplicationMenuItemRepository.getInstance();
        amir.addMenuItem(ApplicationMenuItemRepository.MENUITEM_ADDRESSCARD_VIEW,
        new SampleMenuItem());
    }

    private static class SampleMenuItem extends ApplicationMenuItem {
        SampleMenuItem() {
            super(20);
        }

        public String toString() {
            return "Open the Contacts Demo";
        }

        public Object run(Object context) {
            Contact c = (Contact)context; //an error if this doesn't work
            if ( c != null ) {
                new ContactsDemo().enterEventDispatcher();
            } else {
                throw new IllegalStateException( "Context is null, expected a
   Contact instance");
            }
           net.rim.device.api.ui.component.Dialog.alert("Viewing an email message
   in the email view");
            return null;
        }
    }
}
```

8

Storing persistent data

- Storage options
- Managing persistent data
- Managing custom objects

Storage options

Store data on the handheld in one of the following ways:

- using Mobile Information Device Profile (MIDP) record stores
- using the BlackBerry persistence model

Use the MIDP implementation if you want your application to be portable across multiple devices that are compatible with the Java™ 2 Platform, Micro Edition (J2ME). If you are writing an application specifically for BlackBerry handhelds, use the BlackBerry persistence model because it provides a more flexible and efficient way to store data.

MIDP record store

The `javax.microedition.rms` package provides the MIDP record store implementation. Persistent data is stored in `RecordStore` objects. A record store can be a maximum of 64 KB.

Discrete units of data are called records. A record is an array of bytes that is assigned a unique identification number.

Create a record store

Invoke `openRecordStore()`. Specify `true` to indicate that the record store should be created if it does not exist.

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

Notes: When an application is deleted from a handheld, all record stores created by that application are also deleted.

Each record store has a unique name within a MIDlet suite. A MIDlet can only access record stores that are created by a MIDlet in the same suite.

Add a record

Invoke `addRecord()`.

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

Retrieve a record

To retrieve a record, invoke `getRecord()`. Provide the record ID as parameter.

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

Retrieve all records

Open the record store and then retrieve the enumeration.

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

The `enumerateRecords(RecordFilter filter, RecordComparator comparator, Boolean keepUpdated)` method accepts the following parameters:

Parameter	Description
`filter`	This parameter specifies a `RecordFilter` object to retrieve a subset of record store records (if `null`, all record store records are returned).
`comparator`	This parameter specifies a `RecordComparator` object to determine the order in which records are returned (if `null`, records are returned in an undefined order).
`keepUpdated`	This parameter determines whether the enumeration is kept current with changes to the record store.

BlackBerry persistent storage

There are two main differences between the MIDP record store (`RecordStore`) and the BlackBerry persistence model (`PersistentStore`):

- **Data storage**: MIDP records store data only as byte arrays. In contrast, the BlackBerry APIs enable you to save any object format in the persistent store. As a result, searching for stored data is much faster than in the record model. To store custom object types, the class must implement the `Persistable` interface.
- **Data sharing**: In MIDP, each `RecordStore` belongs to a single MIDlet suite and a MIDlet can only access record stores that are created by a MIDlet in the same suite. In the BlackBerry persistence model, however, data can be shared between applications, at the discretion of the application that creates the data. Code signing ensures that only authorized applications can access the data.

Note: The BlackBerry persistence API is available only with BlackBerry handheld software version 3.6 or later. For earlier versions of handheld software, you must use the MIDP record store.

Conserving storage space

BlackBerry handhelds have limited storage space. You should design your application carefully to minimize the amount of memory that is required to store persistent data.

On a typical BlackBerry handheld, the storage space not required for standard BlackBerry applications must be shared between all applications to store user data, including calendar appointments, contacts, and email messages.

If the handheld is operating with low memory, it might perform the following actions to free memory space:

- delete old email messages from the handheld
- delete calendar appointments that are more than 1 week old from the handheld (if wireless calendar synchronization is enabled)

If the handheld deletes email messages or calendar appointments because of low memory, the data is not deleted from the desktop email program.

Tip: Users can view the current amount of available data storage by clicking **Status** in the handheld options.

Backup and restore

The synchronization API, in the `net.rim.device.api.synchronization` package, enables you to back up and restore custom persistent data on the handheld. See " Adding support for backing up data" on page 94 for more information.

Security

By default, applications on the handheld that have been digitally signed by RIM can access your data in the persistent store. Contact RIM for information on how to control access to your data.

Administrative control

With the BlackBerry Enterprise Server version 3.5 Service Pack 2 or later for Microsoft® Exchange or BlackBerry Enterprise Server version 2.2 or later for IBM® Lotus® Domino®, system administrators can use IT policies to control the use of persistent storage by third-party applications.

Administrators can set the IT policy item `ALLOW_USE_PERSISTENT_STORE` to `TRUE` or `FALSE`. By default, third-party applications are allowed to use persistent storage (`ALLOW_USE_PERSISTENT_STORE` is `TRUE`).

Note: This policy item does not affect the use of the MIDP record store.

Data integrity

To maintain the integrity of data in persistent storage, partial updates are not made if an error occurs during a commit.

Note: Data integrity can be compromised when the VM performs an emergency garbage collection due to low memory. In this case, outstanding transactions are committed immediately. If the handheld fails during this operation, the partially completed transactions are committed when the handheld starts. Outstanding transactions are not committed during normal garbage collection.

Managing persistent data

Persistent data types

A custom data type can be stored persistently if its class implements the `Persistable` interface.

The following native data types can also be stored persistently.

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Object`
- `java.lang.Short`
- `java.lang.String`
- `java.util.Vector`
- `java.util.Hashtable`

Note: When you persist an object, any persistable objects that it refers to are also persisted.

Create a persistent database

Each application typically creates a single `PersistentObject`. This object is the root database of persistent data and indexes for the application. The application saves data into this `PersistentObject`.

Tip: Use a `static` constructor so that the `PersistentObject` is created only once, the first time that an object of this class is created. Each time a process starts, the `static` blocks that it contains are run again.

Each `PersistentObject` is identified by a unique `long` key. This key is typically a hash of the fully qualified package name.

Note: When an application is deleted from a handheld, all persistent objects created by that application are also deleted.

Create a unique long key

1. In the IDE, type a string value, such as **`com.rim.samples.docs.userinfo`**.
2. Select this string.
3. Right-click and click **Convert 'com.rim.samples.docs.userinfo' to long**. The long value appears.

Tip: Include a comment in your code to indicate the string used to generate the long key.

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

Store data persistently

To save data to the persistent store, invoke `setContents()` on a `PersistentObject`. This method replaces existing content with the new content. Invoke `commit()` to save to the persistent store.

Note: If an error occurs during a commit, partial updates are not committed. Data in the `PersistentObject` retains the values from the last commit in order to preserve data integrity.

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

If you have a number of objects that you want to commit to the store, you can commit them in a batch transaction. To do this, invoke `PersistentStore.getSynchObject()` to retrieve the persistent store monitor locking object. Then synchronize on that object, and invoke `commit()` as necessary. When you release the synchronization on the monitor object, all your transactions are committed at once. If any commit in the batch fails, then the entire batch transaction fails. If you invoke `forceCommit()` while synchronized on the monitor object, this object is immediately committed and is not part of the batch transaction.

Retrieve persistent data

Invoke `getContents()` on a `PersistentObject`.

Perform an explicit cast on the object that is returned by `PersistentObject.getContents()` to convert it to your desired format.

```
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]);
   }
}
```

Tip: When an application first accesses a database, it should verify the order of any indexes and recreate the index if a problem exists. Applications should also be able to identify and correct any problems with corrupt or missing data. See " Data integrity" on page 81 for more information.

Delete a database

To delete a database, invoke `PersistentStore.destroyPersistentObject()`. Provide a unique key for the `PersistentObject` as a parameter.

Notes: The `PersistentObject` is used as the root database for the application. By deleting it, you permanently remove all persistent data that the application has stored.

If the .cod file that defines a `PersistentStore` is deleted, all persistent objects created by that .cod are also deleted.

To delete individual data, simply treat the data as normal objects and remove references to it. The data is garbage collected automatically.

Code example

The UserInfo.java example demonstrates how to create an application for users to view their current user name and password, type a new user name and password, and save changes.

Example: UserInfo.java

```
/**
 * UserInfo.java
 * Copyright (C) 2001-2004 Research In Motion Limited. All rights reserved.
 */

package com.rim.samples.docs.userinfo;

import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import java.util.*;
import net.rim.device.api.i18n.*;
import com.rim.samples.docs.baseapp.*;

public class UserInfo extends BaseApp implements UserInfoResource,
        KeyListener, TrackwheelListener {

    private static PersistentObject store;
```

```
private static ResourceBundle _resources;
private AutoTextEditField usernamefield;
private PasswordEditField passwordfield;
private AutoTextEditField currentusernamefield;
private AutoTextEditField currentpasswordfield;

static {
    _resources = ResourceBundle.getBundle(
        UserInfoResource.BUNDLE_ID, UserInfoResource.BUNDLE_NAME);
    store = PersistentStore.getPersistentObject(0xa1a569278238dad2L);
}

private MenuItem saveItem = new MenuItem( _resources.getString(MENUITEM_SAVE),
110, 10) {
    public void run() {
        String username = usernamefield.getText();
        String password = passwordfield.getText();
        String[] userinfo = {username, password};
        synchronized(store) {
            store.setContents(userinfo);
            store.commit();
        }

        Dialog.inform(_resources.getString(APP_SUCCESS));

        usernamefield.setText(null);
        passwordfield.setText(null);
    }
};

private MenuItem getItem = new MenuItem( _resources.getString(MENUITEM_GET),
110, 11 ) {
    public void run() {
        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]);
            }
        }
    }
};

public static void main(String[] args) {
    UserInfo app = new UserInfo();
    app.enterEventDispatcher();
}

public UserInfo() {
    MainScreen mainScreen = new MainScreen();
    mainScreen.setTitle(new LabelField(
            _resources.getString(APPLICATION_TITLE)));

    usernamefield = new AutoTextEditField(
```

```
                _resources.getString(FIELD_NAME), "");
        passwordfield = new PasswordEditField(
            _resources.getString(FIELD_PASSWORD), "");
        currentusernamefield = new AutoTextEditField(
            _resources.getString(FIELD_CURRENTNAME), "");
        currentpasswordfield = new AutoTextEditField(
            _resources.getString(FIELD_CURRENTPASSWORD), "");

        SeparatorField separator = new SeparatorField();

        mainScreen.add(usernamefield);
        mainScreen.add(passwordfield);
        mainScreen.add(separator);
        mainScreen.add(currentusernamefield);
        mainScreen.add(currentpasswordfield);
        mainScreen.addKeyListener(this);
        mainScreen.addTrackwheelListener(this);
        pushScreen(mainScreen);
    }
    public void makeMenu( Menu menu, int instance ) {
        menu.add(saveItem);
        menu.add(getItem);
        super.makeMenu(menu, 0);
    }
    public void onExit() {
        Dialog.alert(_resources.getString(APP_EXIT));
    }
}
```

Managing custom objects

Create a database

Create a `Vector` object to store multiple objects. Create a `PersistentObject` as the root database of your application.

```
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();
      }
   }
}
```

Store data persistently

Objects that implement the `Persistable` interface are persistent.

The following code sample implements the `Persistable` interface as an inner class. It defines an `Object` array with four elements to store a restaurant name, address, phone number, and specialty, and methods to retrieve and set values for `Object` elements.

Note: A class must explicitly implement `Persistable` for objects of the class to be saved persistently. This requirement applies even to subclasses. For example, if class A implements `Persistable`, and it has a subclass B, objects of subclass B cannot be stored persistently unless class B also implements `Persistable`.

```
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;
   }
}
```

Create expandable objects

Use the following strategies to allow you to add fields to your objects:

- Store Boolean values as bits in an `int`. Reserve extra bits for future use.
- Store `Strings` directly, but use a vector or hashtable of key/value pairs so that additional (or seldom-used fields) can be added.
- If you have indexes on a table, store them in a vector or array so that you can add further indexes.

Save an object

Define an object

The following code sample 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

Invoke `addElement()`.

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

Save the updated data

Invoke `setContents()` and `commit()` on the `PersistentObject` to save the updated data.

```
synchronized(store) {
   store.setContents(_data);
   store.commit();
}
```

Tip: Synchronize on the persistent object when you make changes so that other threads cannot make changes to the object at the same time.

Retrieve an object

To 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));
      }
   }
}
```

Code example

The Restaurants.java code example demonstrates how to create an application that enables users to store information about a favorite restaurant.

This code example enables users to save information about multiple restaurants and view information about the most recently saved restaurant.

Example: Restaurants.java

```
/**
 * Restaurants.java
 * Copyright (C) 2004 Research In Motion Limited.
 */

package com.rim.samples.docs.restaurants;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
```

```
import net.rim.device.api.util.*;
import java.util.*;
import net.rim.device.api.i18n.*;
import net.rim.blackberry.api.invoke.*;
import net.rim.blackberry.api.browser.*;
import com.rim.samples.docs.baseapp.*;

public class Restaurants extends BaseApp implements RestaurantResource,
    KeyListener, TrackwheelListener {

    private AutoTextEditField namefield;
    private AutoTextEditField addressfield;
    private EditField phonefield;
    private EditField websitefield;
    private EditField specialtyfield;

    private static Vector _data;
    private static PersistentObject store;
    private static ResourceBundle _resources;

    private MenuItem saveItem = new MenuItem(_resources.getString(MENUITEM_SAVE),
   110, 10) {
        public void run() {
            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.WEBSITE, phonefield.getText());
            info.setElement(RestaurantInfo.SPECIALTY,
            specialtyfield.getText());

            _data.addElement(info);

            synchronized(store) {
                store.setContents(_data);
                store.commit();
            }
            Dialog.inform(_resources.getString(APP_SUCCESS));
            namefield.setText(null);
            addressfield.setText(null);
            phonefield.setText("");
            websitefield.setText("");
            specialtyfield.setText("");
        }
    };

    private MenuItem getItem = new MenuItem(_resources.getString(MENUITEM_GET),
   110, 11) {
        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));
                websitefield.setText(info.getElement(RestaurantInfo.WEBSITE));

specialtyfield.setText(info.getElement(RestaurantInfo.SPECIALTY));
            }
        }
    }
};

private MenuItem phoneItem = new MenuItem(_resources.getString(MENUITEM_PHONE),
110, 12) {
    public void run() {
        synchronized(store) {
            String phoneNumber = phonefield.getText();
            if ( phoneNumber.length() == 0) {
                Dialog.alert(_resources.getString(ALERT_NO_PHONENUMBER));
            } else {
                PhoneArguments call = new
PhoneArguments(PhoneArguments.ARG_CALL, phoneNumber);
                Invoke.invokeApplication(Invoke.APP_TYPE_PHONE, call);
            }
        }
    }
};
private MenuItem browserItem = new
MenuItem(_resources.getString(MENUITEM_BROWSER), 110, 12) {
    public void run() {
        synchronized(store) {
            String websiteUrl = websitefield.getText();
            if (websiteUrl.length() == 0) {
                Dialog.alert(_resources.getString(ALERT_NO_WEBSITE));
            } else {
                BrowserSession visit = Browser.getDefaultSession();
                visit.displayPage(websiteUrl);
            }
        }
    }
};
static {
    _resources = ResourceBundle.getBundle(
    RestaurantResource.BUNDLE_ID,
    RestaurantResource.BUNDLE_NAME);
    store = PersistentStore.getPersistentObject(0xdec6a67096f833cL);
    // Key is hash of test.samples.restaurants.
    synchronized (store) {
        _data = (Vector)store.getContents();
        if (_data == null) {
            _data = new Vector();
            store.setContents( _data );
            store.commit();
        }
    }

}

public static void main(String[] args) {
```

```
        Restaurants app = new Restaurants();
        app.enterEventDispatcher();
    }
    private static final class RestaurantInfo implements Persistable {

        // Data.
        private String[] _elements;

        // Fields.
        public static final int NAME = 0;
        public static final int ADDRESS = 1;
        public static final int PHONE = 2;
        public static final int WEBSITE = 3;
        public static final int SPECIALTY = 4;

        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;
        }
    }

    public Restaurants() {
        MainScreen mainScreen = new MainScreen();
        mainScreen.setTitle(new LabelField(
        _resources.getString(APPLICATION_TITLE)));
        namefield = new AutoTextEditField(
        _resources.getString(FIELD_NAME), "");
        addressfield = new AutoTextEditField(
        _resources.getString(FIELD_ADDRESS), "");
        phonefield = new EditField(
        _resources.getString(FIELD_PHONE), "", Integer.MAX_VALUE,
        BasicEditField.FILTER_PHONE);
        websitefield = new EditField(
        _resources.getString(FIELD_WEBSITE), "", Integer.MAX_VALUE,
        BasicEditField.FILTER_URL);
        specialtyfield = new EditField(
        _resources.getString(FIELD_SPECIALTY), "",
        Integer.MAX_VALUE, BasicEditField.FILTER_DEFAULT);

        mainScreen.add(namefield);
        mainScreen.add(addressfield);
        mainScreen.add(phonefield);
        mainScreen.add(websitefield);
        mainScreen.add(specialtyfield);
        mainScreen.addKeyListener(this);
```

```
        mainScreen.addTrackwheelListener(this);
        pushScreen(mainScreen);
    }

    public void makeMenu( Menu menu, int instance ) {
        menu.add(saveItem);
        menu.add(getItem);
        menu.add(phoneItem);
        menu.add(browserItem);
        super.makeMenu(menu, instance);
    }

    public void onExit() {
        Dialog.alert(_resources.getString(APP_EXIT));
    }
}
```

9

Backing up and restoring persistent data

- Synchronization API
- Adding support for backing up data

Synchronization API

The synchronization API, in the `net.rim.device.api.synchronization` package, enables your application to integrate with the BlackBerry Desktop Software to perform two tasks:

- back up a database to a desktop file and restore it later
- synchronize data with a desktop application

Tip: The BlackBerry Desktop Software requires that backup data use the following format:

```
Length<2> Type<1> Data<n>
```

To ensure that the data is in the appropriate format, use one of the write methods in the `net.rim.device.api.synchronization.ConverterUtilities` class.

Data backup

The BlackBerry Desktop Software provides a Backup and Restore tool that enables users to save handheld data to a file on their desktop, and use this file to restore data to the handheld.

When an application implements the synchronization API, the desktop software backs up and restores the application database along with the other handheld databases. You can also use the synchronization API to create data archives or to populate application databases when the handheld first connects to the computer.

Data synchronization

The desktop software provides an **Intellisync** tool to synchronize the handheld with the applications on the user computer.

Where backup and restore performs a bulk load of data between the handheld and a desktop backup file, synchronization compares the data that exists in a desktop application with the data on the handheld to merge the data.

To synchronize data with a desktop application, write a plug-in for the desktop software using the BlackBerry Desktop API. The BlackBerry JDE also includes a sample synchronization application with a desktop plug-in.

Tip: There are no restrictions on the format that you use to store data for backup. The only requirement is that your handheld application must read and write data in the same format that is used by your desktop plug-in application.

Synchronization API overview

Implement the following interfaces provided by the synchronization API:

Interface	Description
SyncConverter	converts data between the SyncObject format required on the handheld and a serialized format required on the desktop
SyncCollection	represents the collection of synchronization objects for an application
SyncObject	represents an object that can be backed up and restored to the user computer

The SerialSyncManager class provides access to the handheld synchronization manager, in particular to register new objects for synchronization.

Note: To back up and restore a very small amount of data, such as application configuration options, you can extend the SyncItem class and implement its abstract methods. The SyncItem class implements the SyncCollection, SyncConverter, and SyncObject interfaces for you.

Adding support for backing up data

To support backup, modify a class that implements the Persistable interface to implement the SyncObject interface.

Modify the main class for your application to implement the SyncCollection and SyncConverter interfaces.

Note: The SyncCollection and SyncConverter interfaces can be implemented by the same class or by separate classes, depending on the design of your application. The following sections explain how to implement these interfaces in the same class.

Define a unique ID

Define a _uid variable and implement getUID() to return a unique ID to use for synchronization operations.

Define a constructor

Define a constructor that accepts a unique ID as a parameter and sets the _uid variable to this value.

Note: Each synchronization object that is stored on the handheld must have an associated ID that is unique to its application. The UIDGenerator sets this value ID by default.

Register a synchronization collection

In main(), register your SyncCollection with the handheld synchronization manager. Create a separate project to pass in the init argument when the handheld first starts. See " Create an initialization project" on page 95 for more information.

```
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();
    }
}
```

Create an initialization project

To register a synchronization collection when the handheld starts, create a separate project that acts as an alternate entry point to your main application. This passes an argument to your application the first time that the handheld starts so that your application registers only once.

1. In the IDE, create a project.
2. Right-click the project and click **Properties**.
3. Click the **Application** tab.
4. In the **Project type** drop-down list, click **Alternate CLDC Application Entry Point**.
5. In the **Alternate entry point for** drop-down list, click the project that implements synchronization.
6. In the **Arguments passed to** field, type init.
7. Select the **Auto-run on startup** option.
8. Select the **System module** option.
9. Click **OK**.

Note: Arguments can be passed to BlackBerry CLDC applications on startup, however, this functionality does not exist for MIDLet applications.

Code example

The RestaurantsSync.java code example demonstrates how to enable desktop software to back up and restore persistent data for your application. This example modifies the Restaurants.java code example to implement the synchronization API.

Example: RestaurantsSync.java

```
/**
 * RestaurantsSync.java
 * Copyright (C) 2001-2004 Research In Motion Limited. All rights reserved.
 */

package com.rim.samples.docs.restaurantssync;

import java.io.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
```

```
import java.util.*;
import net.rim.device.api.i18n.*;
import net.rim.device.api.synchronization.*;

import com.rim.samples.docs.baseapp.*;

public class RestaurantsSync extends BaseApp implements RestaurantsSyncResource,
        SyncCollection, SyncConverter, KeyListener, TrackwheelListener {

    private static final long KEY = 0xdec6a67096f833cL;

    private AutoTextEditField namefield;
    private AutoTextEditField addressfield;
    private EditField phonefield;
    private EditField specialtyfield;

    private static PersistentObject store;
    private static Vector _data;
    private static ResourceBundle _resources;
    private static final int FIELDTAG_NAME = 1;
    private static final int FIELDTAG_PHONE = 2;
    private static final int FIELDTAG_ADDRESS = 3;
    private static final int FIELDTAG_SPECIALTY = 4;

    private static RestaurantsSync _instance;

   private MenuItem saveItem = new MenuItem(_resources, MenuItem.SAVE_CLOSE, 110,
   10) {
        public void run() {
            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());
            _data.addElement(info);

            synchronized(store) {
                store.setContents(_data);
                store.commit();
            }
            Dialog.inform(_resources.getString(APP_SUCCESS));
            namefield.setText(null);
            addressfield.setText(null);
            phonefield.setText("");
            specialtyfield.setText("");
        }
    };
    private MenuItem getItem = new MenuItem("Get", 110, 11) {
        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));
            }
        }
    }
};

static {
    _resources = ResourceBundle.getBundle(RestaurantsSyncResource.BUNDLE_ID,
            RestaurantsSyncResource.BUNDLE_NAME);
    store = PersistentStore.getPersistentObject(KEY);
    synchronized (store) {
        _data = (Vector)store.getContents();
        if ( _data == null ) {
            _data = new Vector();
            store.setContents( _data );
            store.commit();
        }
    }

}
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.
        SyncManager.getInstance().enableSynchronization(
                RestaurantsSync.getInstance());
    } else {
        RestaurantsSync app = new RestaurantsSync();
        app.enterEventDispatcher();
    }
}

public static RestaurantsSync getInstance() {
    if (_instance == null) {
        _instance = new RestaurantsSync();
    }
    return _instance;
}

private static final class RestaurantInfo implements Persistable, SyncObject {
    private String[] _elements; // Data.
    public static final int NAME = 0;
    public static final int ADDRESS = 1;
    public static final int PHONE  = 2;
    public static final int SPECIALTY = 3;
    private int _uid;

    public int getUID() {
```

```
            return _uid;
        }

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

        public String getElement(int id) {
            return _elements[id];
        }

        public void setElement(int id, String value) {
            _elements[id] = value;
        }
    }

    // SyncConverter methods.
    public SyncObject convert(DataBuffer data, int version, int UID) {
        try {
            RestaurantInfo info = new RestaurantInfo(UID);
            while(data.available() > 0) {
                int length = data.readShort();
                byte[] bytes = new byte[length];
                switch (data.readByte()) {
                    case FIELDTAG_NAME:
                        data.readFully(bytes);
                        //trim null-terminator
                        info.setElement(RestaurantInfo.NAME,
                                new String(bytes).trim());
                        break;
                    case FIELDTAG_PHONE:
                        data.readFully(bytes);
                        info.setElement(RestaurantInfo.PHONE,
                                new String(bytes).trim());
                        break;
                    case FIELDTAG_ADDRESS:
                        data.readFully(bytes);
                        info.setElement(RestaurantInfo.ADDRESS,
                                new String(bytes).trim());
                        break;
                    case FIELDTAG_SPECIALTY:
                        data.readFully(bytes);
                        info.setElement(RestaurantInfo.SPECIALTY,
                                new String(bytes).trim());
                        break;
```

```
                default:
                    data.readFully(bytes);
                    break;
            }
        }
        return info;
    } catch (EOFException e) {
        System.err.println(e.toString());
    }
    return null;
}

public boolean convert(SyncObject object, DataBuffer buffer, int version) {
    if (version == getSyncVersion()) {
        if (object instanceof RestaurantInfo )
        {
            String name = ((RestaurantInfo)object).getElement(
                    RestaurantInfo.NAME);
            String phone = ((RestaurantInfo)object).getElement(
                    RestaurantInfo.PHONE);
            String address = ((RestaurantInfo)object).getElement(
                    RestaurantInfo.ADDRESS);
            String specialty = ((RestaurantInfo)object).getElement(
                    RestaurantInfo.SPECIALTY);
            buffer.writeShort(name.length()+1);
            buffer.writeByte(FIELDTAG_NAME);
            buffer.write(name.getBytes());
            buffer.writeByte(0);
            buffer.writeShort(phone.length()+1);
            buffer.writeByte(FIELDTAG_PHONE);
            buffer.write(phone.getBytes());
            buffer.writeByte(0);
            buffer.writeShort(address.length()+1);
            buffer.writeByte(FIELDTAG_ADDRESS);
            buffer.write(address.getBytes());
            buffer.writeByte(0);
            buffer.writeShort(specialty.length()+1);
            buffer.writeByte(FIELDTAG_SPECIALTY);
            buffer.write(specialty.getBytes());
            buffer.writeByte(0);
            return true;
        }
    }
    return false;
}
public void beginTransaction() {
    store = PersistentStore.getPersistentObject(KEY);
    _data = (Vector)store.getContents();
}
public void endTransaction() {
    store.setContents(_data);
    store.commit();
}
public SyncConverter getSyncConverter() {
    return this;
```

```
    }
    public String getSyncName() {
        return "Restaurant Synchronization Demo";
    }
    public String getSyncName(Locale locale) {
        return getSyncName();
    }
    public int getSyncObjectCount() {
        store = PersistentStore.getPersistentObject(KEY);
        _data = (Vector)store.getContents();
        return _data.size();
    }
    public SyncObject[] getSyncObjects() {
        SyncObject[] array = new SyncObject[_data.size()];
        for (int i = _data.size() - 1; i >= 0; --i) {
            array[i] = (SyncObject)_data.elementAt(i);
        }
        return array;
    }
    public SyncObject getSyncObject(int uid) {
        for (int i = _data.size() -1; i>= 0; --i) {
            SyncObject so = (SyncObject)_data.elementAt(i);
            if (so.getUID() == uid ) return so;
        }
        return null;
    }
    public int getSyncVersion() {
        return 1;
    }
    public boolean addSyncObject(SyncObject object) {
        _data.addElement(object);
        return true;
    }
    public boolean removeAllSyncObjects() {
        _data.removeAllElements();
        return true;
    }
    public void clearSyncObjectDirty(SyncObject object) {
        // Not applicable.
    }
    public boolean isSyncObjectDirty(SyncObject object) {
        return false;
    }
    public boolean removeSyncObject(SyncObject object) {
        return false;
    }
    public void setSyncObjectDirty(SyncObject object) {
    }
    public boolean updateSyncObject(SyncObject oldObject, SyncObject newObject) {
        return false;
    }
    public RestaurantsSync() {
        MainScreen mainScreen = new MainScreen();
        mainScreen.setTitle(new LabelField(
_resources.getString(APPLICATION_TITLE)));
        namefield = new AutoTextEditField(_resources.getString(FIELD_NAME), "");
```

```
        addressfield = new AutoTextEditField( _resources.getString(FIELD_ADDRESS),
    "");
        phonefield = new EditField(
    _resources.getString(FIELD_PHONE), "", Integer.MAX_VALUE,
    BasicEditField.FILTER_PHONE);
        specialtyfield = new EditField(_resources.getString(FIELD_SPECIALTY), "",
    Integer.MAX_VALUE, BasicEditField.FILTER_DEFAULT);
        mainScreen.add(namefield);
        mainScreen.add(addressfield);
        mainScreen.add(phonefield);
        mainScreen.add(specialtyfield);
        mainScreen.addKeyListener(this);
        mainScreen.addTrackwheelListener(this);
        pushScreen(mainScreen);
    }
    public void makeMenu( Menu menu, int instance) {
        menu.add(saveItem);
        menu.add(getItem);
        super.makeMenu(menu, instance);
    }
    public void onExit() {
        Dialog.alert(_resources.getString(APP_EXIT));
    }
}
```

10

Accessing setup and configuration information

- Service book API

Service book API

The service book API (`net.rim.device.api.servicebook`) enables handheld applications to interact with the BlackBerry infrastructure. The service book consists of service records, each of which defines a service that can be enabled on a handheld.

Service records define the communication protocol (WAP or IPPP), network gateway, and configuration information such as browser settings.

The service book API enables applications to perform the following functions:

- **Manage Mobile Data Service connections:** The browser application API can connect to the wireless network using any `ServiceBook` entry with a `UID` of `BrowserConfig`. For example, the `Browser` class uses the service book to retrieve a `BrowserSession`. `Browser.getTransportUid()` queries the service book to retrieve the UID associated with a given service record.
- **Manage mail information:** The mail API enables applications to specify a channel through which to send e-mail by referencing the appropriate service record. For example, applications can choose to send email via a BlackBerry Enterprise Server or a BlackBerry Internet Email Service. See " BlackBerry mail API" on page 11 for more information.

To view the service book on handhelds, users click **Service Book** under the handheld options.

The `ServiceBook` class maintains a collection of `ServiceRecord` objects. Each `ServiceRecord` object is identified by a unique ID (`UID`) and connection ID (`CID`).

CID	Description
`CMIME`	The compressed multipurpose mail extensions (CMIME) CID defines email connections.
`ALP`	The address lookup protocol (ALP) CID defines connections for wireless Global Address List searches.
`IPPP`	The IP Proxy Protocol (IPPP) CID defines HTTP connections using Mobile Data Service.
`BrowserConfig`	The browser configuration (BrowserConfig) CID defines BlackBerry and WAP browser connections.
`Sync`	The data synchronization (Sync) CID defines connections for wireless data synchronization.
`WAP`	The wireless application protocol (WAP) CID defines WAP gateway connections.
`CICAL`	The compressed iCalendar (CICAL) CID defines connections for wireless calendar synchronization.

Service record	Description
Desktop [CMIME]	This service record contains information required to send email messages using the desktop and perform other functions, such as wireless email reconciliation.
Desktop [ALP]	This service record contains information required to perform wireless Global Address Book searches.
Desktop [IPPP]	This service record contains information that is required to use and browse the Internet using Mobile Data Service.
Desktop [CICAL]	This service record contains information that is required to perform wireless calendar operations.

Service record	Description
Desktop [BrowserConfig]	This service record contains configuration information for the BlackBerry Browser.
Web Client [CMIME]	This service record contains information required to send messages and perform functions (for example, wireless email reconciliation) using the BlackBerry Internet Email Service.
WAP Secure Transport [WAP]	This service record contains information that is required to connect to a service provider's Wireless Application Protocol (WAP) gateway.
WAP Browser [BrowserConfig]	This service record contains configuration information for the WAP Browser.
Desktop [Sync]	This service record contains information that is required to perform data synchronization.

Listen for service book events

Implement the `GlobalEventListener` interface (in the `net.rim.device.api.system` package). To specify the actions to perform when a global event is received, implement `GlobalEventListener.eventOccurred()`.

To register the global event listener, invoke `Application.addGlobalEventListener(GlobalEventListener)`.

The `ServiceBook` class defines the following global events, identified by global unique identifiers (GUID).

GUID	Description
GUID_SB_ADDED	The GUID for the global event that is sent when a service book is added.
GUID_SB_BR_END	The GUID for the global event that is sent when service book backup-restore ends.
GUID_SB_BR_START	The GUID for the global event that is sent when service book backup-restore starts.
GUID_SB_CHANGED	The GUID for the global event that is sent when a service book is changed.
GUID_SB_OTA_SWITCH	The GUID for the global event that is sent when all service records are inserted due to a move BlackBerry Enterprise Server command over the air.
GUID_SB_OTA_UPDATE	The GUID for the global event that is sent when all service records are updated for a UID over the air.
GUID_SB_REMOVED	The GUID for the global event that is sent when a service book is deleted.

11

Managing notifications

- Notification API
- Adding an event
- Responding to events
- Customizing system notifications

Notification API

The notification API (`net.rim.device.api.notification`) enables you to add custom events for your application and define the type of notification that users receive when custom events occur.

Note: Check for a `ControlledAccessException` when your application first accesses the notification API. This runtime exception is thrown if the system administrator restricts access to the notification API using application control. See "Application control" on page 12 of the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information.

There are two types of notification events:

- **Immediate events**: system notification, such as flashing LED, vibration, or tune
- **Deferred events**: application-specific notification, such as a user interface

With immediate events, the handheld provides notification to the user as soon as the event occurs, using a system notification, such as a flashing LED, vibration, or tune. An application cannot request a specific type of notification. In the handheld profiles list, users control how they receive notification of immediate events by choosing an active profile and setting profile options. To add custom system notifications for immediate events, implement the `Consequence` interface.

With deferred events, the handheld schedules events in a queue according to their priority. When the event occurs, applications that are affected by the event can provide a custom notification to the user, typically by displaying a user interface element such as a dialog box. To listen for deferred events, implement the `NotificationsEngineListener` interface. The handheld does not provide system-wide notification for deferred events.

Adding events

Register a new event source

Create a unique long ID

Define a `long` ID for each notification event.

```
public static final long ID_1 = 0xdc5bf2f81374095L;
```

Tip: Use the IDE to convert a `String` to a `long` to create a similar identifier for your application:

1. In the IDE text pane, type a string.
2. Select the string, right-click and click **Convert "*string*" to Long**.

Define a source object

Define an object that provides the source for the event. This object must implement `toString()`, which returns the string to display in the profiles list.

```
Object event = new Object() {
    public String toString() {
        return "Notification Demo";
    }
}
```

Register your application as a notification source

To add your application to the handheld profiles list as the source of an event, invoke `NotificationsManager.registerSource()`. In this method, specify a unique event ID, the source object, and the notification level.

Notification level sets the priority of the event, which determines the order in which deferred events occur. The levels, in order from highest to lowest priority, are as follows:

- `NotificationsConstants.CRITICAL`
- `NotificationsConstants.SENSITIVE`
- `NotificationsConstants.IMPORTANT`
- `NotificationsConstants.DEFAULT_LEVEL`
- `NotificationsConstants.CASUAL`

Note: The priority level applies to deferred events only. Immediate events occur as soon as they are triggered.

When you trigger a deferred event, you specify an expiry time. The user might not receive notification of a lower-priority event, if the event expires before higher-priority events complete.

Create a library project

To register an event source, create a library project with `libMain()` to perform the registration when the handheld starts.

1. In the IDE, create a project.
2. Right-click the project and click **Properties**.
3. Click the **Application** tab.
4. In the **Project type** drop-down list, click **Library**.
5. Select the **Auto-run on startup** option.
6. Click **OK**.
7. Define `libMain()`.

```
public static final long ID_1 = 0xdc5bf2f81374095L;
public static final Object event = new Object() {
    public String toString() { return "Sample Notification Event #1"; }
};
public static void libMain(String[] args) {
    NotificationsManager.registerSource(ID_1, event,
    NotificationsConstants.CASUAL);
}
```

Trigger an immediate event

Invoke `triggerImmediateEvent()`. Immediate events are indicated by standard system notifications, such as tune, vibration, or LED.

```
NotificationsManager.triggerImmediateEvent(ID_1, 0, this, null);
```

The `triggerImmediateEvent` method accepts the following parameters:

Parameter	Description
`sourceID`	identifier of the application that starts the event (as specified when you invoked `registerSource()`)
`eventID`	application-specific event identifier
`eventReference`	application-specific event cookie
`context`	optional context object

In most cases, do not use immediate events because the handheld event notification does not adequately indicate to the user what has happened. For example, if the handheld vibrates, it would be difficult for the user to know whether an event has occurred in your application, or whether a new email message has arrived. If you do use immediate events, consider implementing a custom notification, such as a particular tune, to distinguish your application events from other handheld events. See " Customizing system notifications" on page 112 for more information.

Trigger a deferred event

Invoke `negotiateDeferredEvent()`. A deferred event enables your application to notify the user with a user interface element, such as a dialog box.

```
NotificationsManager.negotiateDeferredEvent(ID_1, 0, this, -1,
NotificationsConstants.MANUAL_TRIGGER, null);
```

The `negotiateDeferredEvent()` method accepts the following parameters:

Parameter	Description
`sourceID`	identifier of the application that starts the event (as specified when you invoked `registerSource()`)
`eventID`	application-specific event identifier
`eventReference`	application-specific event cookie
`timeout`	event expiry time, in milliseconds, relative to time when the method is invoked (the timeout is ignored unless the trigger is `OUT_OF_HOLSTER_TRIGGER`)
`trigger`	either `NotificationsConstants.OUT_OF_HOLSTER_TRIGGER`, which specifies that the event occurs when the handheld is disconnected from the computer; or `NotificationsConstants.MANUAL_TRIGGER`, which specifies that the application itself triggers this event
`context`	optional object that can store additional, arbitrary parameters to control the state or behavior of an event notification

If you invoke `negotiateDeferredEvent()`, your application must implement the `NotificationEventListener` to receive events and respond appropriately. See " Responding to events" on page 110 for more information.

Cancel an event

Cancel an immediate event

Invoke `cancelImmediateEvent()`, and specify the source and event IDs.

```
NotificationsManager.cancelImmediateEvent(ID_1, 0, this, null);
```

Cancel a deferred event

Invoke `cancelDeferredEvent()` and specify the source and event ID.

```
NotificationsManager.cancelDeferredEvent(ID_1, 0, this,
   NotificationsConstants.MANUAL_TRIGGER, null);
```

Cancel all deferred events

Invoke `cancelAllDeferredEvents()` to cancel all deferred events that your application started.

```
NotificationsManager.cancelAllDeferredEvents(ID_1,
   NotificationsConstants.MANUAL_TRIGGER, null);
```

Tip: If you invoke `negotiateDeferredEvent()` and do not specify a timeout, you must invoke `cancelDeferredEvent()` to cancel the event, or the event never expires.

Code example

Example: NotificationDemo.java

```
/**
 * NotificationsDemo.java
 * Copyright (C) 2001-2004 Research In Motion Limited. All rights reserved.
 */

package com.rim.samples.docs.notifications;

import net.rim.device.api.notification.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import com.rim.samples.docs.baseapp.*;

public class NotificationsDemo extends BaseApp {

    public static final long ID_1 = 0xdc5bf2f81374095L;
    private long _eventIdGenerator;
    private static Object er;

    public static final Object event = new Object() {
        public String toString() {
            return "Sample Notification Event #1";
        }
    };

    public static void main(String[] args) {
        NotificationsManager.registerSource(ID_1, event,
   NotificationsConstants.CASUAL);
        NotificationsManager.registerConsequence(ConsequenceDemo.ID, new
   ConsequenceDemo());
        NotificationsDemo app = new NotificationsDemo();
```

```
    app.enterEventDispatcher();
}

public NotificationsDemo() {
    MainScreen mainScreen = new MainScreen();
    mainScreen.setTitle("Notification Demo App");
    mainScreen.addKeyListener(this);
    mainScreen.addTrackwheelListener(this);
    NotificationsManager.registerNotificationsEngineListener(ID_1,
        new NotificationsEngineListenerImpl(this));
    pushScreen(mainScreen);
}

private MenuItem triggerItem = new MenuItem(null, 0, 100, 10) {
    public void run() {
        NotificationsManager.triggerImmediateEvent(ID_1, 0, this, null);
    }
    public String toString() {
        return "Trigger event";
    }
};

private MenuItem deferItem = new MenuItem(null, 0, 100, 10) {
    public void run() {
        long timeout = -1; // Ignored unless trigger is OUT_OF_HOLSTER_TRIGGER.
        int trigger = NotificationsConstants.MANUAL_TRIGGER;
        Object er = new Object();
       NotificationsManager.negotiateDeferredEvent(ID_1, ++_eventIdGenerator,
                er, timeout, trigger, null);
            }
    public String toString() {
        return "Start deferred event";
    }
};
private MenuItem cancelItem = new MenuItem(null, 0, 100, 10) {
    public void run() {
        int trigger = NotificationsConstants.MANUAL_TRIGGER;
        NotificationsManager.cancelDeferredEvent(ID_1, _eventIdGenerator, er,
            trigger, null);
        }
    public String toString() {
        return "Cancel deferred event";
    }
};

public void makeMenu( Menu menu, int instance ) {
    menu.add(triggerItem);
    menu.add(deferItem);
    menu.add(cancelItem);
    super.makeMenu(menu, instance);
}

public void onExit() {
    System.exit(0);
}
```

```
    private static class NotificationsEngineListenerImpl implements
            NotificationsEngineListener {
        private UiApplication _app;
        public NotificationsEngineListenerImpl(UiApplication app) {
            _app = app;
        }

        public void deferredEventWasSuperseded(long sourceID, long eventID,
                Object eventReference, Object context) {
            final long _eventID = eventID;
            er = eventReference;
            _app.invokeLater(new Runnable() {
                public void run() {
                    NotificationsManager.cancelDeferredEvent(ID_1, _eventID, er,
                            NotificationsConstants.MANUAL_TRIGGER, null);
                }
            });
        }
        public void notificationsEngineStateChanged(int stateInt, long sourceID,
                long eventID, Object eventReference, Object context) {
            if(stateInt == NotificationsConstants.OUT_OF_HOLSTER_ENGINE_STATE) {
                // Perform some action if handheld is removed from holster.
            }
            if(stateInt == NotificationsConstants.IN_HOLSTER_ENGINE_STATE) {
                // Perform some action if handheld is inserted into holster.
            }
        }
        public void proceedWithDeferredEvent(long sourceID, long eventID,
                Object eventReference, Object context) {
            final long _eventID = eventID;
            _app.invokeLater(new Runnable() {
                public void run() {
                    String s = "This event has occurred:  " + _eventID;
                    Dialog d = new Dialog(Dialog.D_OK, s, Dialog.OK,
                            Bitmap.getPredefinedBitmap(Bitmap.INFORMATION), 0);
                    d.show();
                }
            });
        }
    }
}
```

Responding to events

To define custom notification, implement `NotificationsEngineListener` and then register it by invoking `negotiateDeferredEvent()`. You do not need to implement the listener if you trigger immediate events, for which the handheld provides standard system notification.

Provide a custom UI notification for deferred events

Implement the `NotificationsEngineListener` interface. See the *BlackBerry Application Developer Guide Volume 1: Fundamentals* for more information on creating user interfaces.

```
private static class ListenerImpl implements NotificationsEngineListener {...}
```

Define behaviour if an event is superseded

Implement `deferredEventWasSuperseded()`. This method is invoked when the event is superseded by another event at the same or higher priority level. For example, you could cancel the event if it is superseded.

```
public void deferredEventWasSuperseded(long sourceID, long eventID, Object
   eventReference, Object context) {
      final long _eventID = eventID;
      er = eventReference;
      _app.invokeLater(new Runnable() {
      public void run() {
         NotificationsManager.cancelDeferredEvent(ID_1, _eventID, er,
         NotificationsConstants.MANUAL_TRIGGER, null);
      }
   });
}
```

Define holstering behaviour

Implement `notificationsEngineStateChanged()`. This method is invoked when the handheld is inserted in, or removed from, the holster. For example, you could perform a specific action when a deferred event is scheduled and the handheld is connected to, or disconnected from, the computer.

```
public void notificationsEngineStateChanged(int stateInt, long sourceID, long
   eventID, Object eventReference, Object context) {
   if(stateInt == otificationsConstants.OUT_OF_HOLSTER_ENGINE_STATE) {
      // perform action if handheld is removed from holster
   }
   if(stateInt == NotificationsConstants.IN_HOLSTER_ENGINE_STATE) {
      // perform action if handheld is inserted into holster
   }
}
```

Define notification

Implement `proceedWithDeferredEvent()` to define how to notify the user when the event occurs, such as by displaying a dialog box. This method is invoked when the listener proceeds with the event (no other higher priority events are in the queue).

```
public void proceedWithDeferredEvent(long sourceID, long eventID, Object
   eventReference, Object context) {
      final long _eventID = eventID;
      _app.invokeLater(new Runnable() {
         public void run() {
         String s = "This event has occurred:  " + _eventID;
         Dialog d = new Dialog(Dialog.D_OK, s, Dialog.OK,
         Bitmap.getPredefinedBitmap(Bitmap.INFORMATION), 0);
         d.show();
```

```
            _eventHashtable.put(_eventID, d);
        }
    });
}
```

Register the notifications listener

Register your listener with the `NotificationsManager`. Provide as parameters the event source ID of your application and an instance of the class that implements the `NotificationsEngineListener` interface.

```
NotificationsManager.registerNotificationsEngineListener( ID_1, new
    ListenerImpl(this));
```

Note: You can only register one `NotificationsEngineListener` for each application.

Customizing system notifications

Implement the `Consequence` interface to create custom system notifications, such as a particular tune, for immediate events or to perform other actions when events occur, such as creating a log that counts the number of notifications received.

Note: The `Consequence` interface is used only for immediate events that require system notification. Deferred events require your application to implement the `NotificationsEngineListener` and respond with an application-specific response.

Provide an application on the handheld Home screen to enable users to set notification options.

Respond to notification events

Implement the `Consequence` and `SyncConverter` interfaces to respond to notification events. The `Consequence` interface defines an application response to notification events. The `SyncConverter` interface defines the functionality required to convert data from object to serialized format. It is required to enable the handheld to back up and restore profile configurations. See " Backing up and restoring persistent data" on page 89 for more information.

```
private static class ConsequenceImpl implements Consequence,SyncConverter {...}
```

Define a unique ID

Define a unique ID for this consequence.

```
public static final long ID = 0xbd2350c0dfda2a51L;
```

Define constants

Declare `DATA` and `TYPE` constants for the application. These constants are used when identifying the type of incoming data from the `SyncConverter` when `convert()` is invoked. They are arbitrary identifiers for data that is appropriate to this application.

```
private static final int TYPE = 'n' << 24 | 'o' << 16 | 't' << 8 | 'd';
```

```
private static final byte[] DATA = new byte[] {'m', 'y', '-', 'c',
'o', 'n', 'f', 'i', 'g', '-', 'o', 'b', 'j', 'e', 'c', 't'};
private static final Configuration CONFIG = new Configuration(DATA);
```

Create a tune

Create a tune to be played as part of the consequence for event notifications.

```
private static final short BFlat = 466; //466.16
private static final short TEMPO = 125;
private static final short d16 = 1 * TEMPO;
private static final short dpause = 10; //10 millisecond pause
private static final short[] TUNE = new short[] {BFlat, d16, pause, BFlat};
private static final int VOLUME = 80; //percentage volume
```

Play a tune from a supported audio format

On handhelds that support standard audio formats, you can also play back an audio file in one of the following supported formats:

- audio/adpcm
- audio/midi
- audio/x-midi
- audio/mid

BlackBerry handhelds use the Mobile Media API (`javax.microedition.media`) to support standard audio formats.

To determine the supported audio formats at runtime, invoke `Manager.getSupportedContentTypes()`.

See the `javax.microedition.media` package in the *API Reference* for information.

Define a notification

Implement `startNotification()` to define the notification for this consequence. In the following code sample, the LED flashes and a tune is played.

```
public void startNotification(long consequenceID, long sourceID, long eventID,
   Object configuration, Object context) {
   LED.setConfiguration(500, 250, LED.BRIGHTNESS_50);
   LED.setState(LED.STATE_BLINKING);
   Alert.startAudio(TUNE, VOLUME);
   Alert.startBuzzer(TUNE, VOLUME);
}
```

Stop a notification

Implement `stopNotification()` to stop notification for this consequence.

```
public void stopNotification(long consequenceID, long sourceID,
   long eventID, Object configuration, Object context) {
```

```
    LED.setState(LED.STATE_OFF);
    Alert.stopAudio();
    Alert.stopBuzzer();
}
```

Store user profile settings

Implement `newConfiguration()` to create a new configuration object that stores user profile settings. This object is passed to the consequence implementation to determine whether the type of consequence that the user specified is appropriate for the event. The following code sample returns the `CONFIG` object that you defined earlier.

```
public Object newConfiguration(long consequenceID, long sourceID,
byte profileIndex, int level, Object context) {
    return CONFIG;
}
```

Enable handheld data backup

Implement `SyncConverter.convert()`. This method is invoked when data is backed up from the handheld to the user computer. The following sample implementation reads incoming data from the `DataBuffer` and applies a four-byte type and length to the raw data.

```
public SyncObject convert(DataBuffer data, int version, int UID) {
    try {
        int type = data.readInt();
        int length = data.readCompressedInt();
        if ( type == TYPE ) {
            byte[] rawdata = new byte[length];
            data.readFully(rawdata);
            return new Configuration(rawdata);
        }
    } catch (EOFException e) {
        System.err.println(e);
    }
    return null;
}
```

Enable handheld data restore

Implement `SyncConverter.convert()`. This method is invoked when data is restored from the user computer to the handheld.

```
public boolean convert(SyncObject object, DataBuffer buffer, int version) {
    boolean retval = false;
    if ( object instanceof Configuration ) {
        Configuration c = (Configuration)object;
        buffer.writeInt(TYPE);
        buffer.writeCompressedInt(c._data.length);
        buffer.write(c._data);
        retval = true;
    }
```

```
    return retval;
}
```

Define the notification configuration

Create a class that describes the notification configuration information. The class implements `SyncObject` and `Persistable`. You must implement `SyncObject.getUID()` but you can return 0 because this value is required only for data synchronization, which is not required in this example.

```
private static final class Configuration implements SyncObject, Persistable {
    public byte[] _data;
    public Configuration(byte[] data) {
        _data = data;
    }
    public int getUID() {
        return 0;
    }
}
```

Register a consequence

If you create a custom `Consequence` implementation, register it with the `NotificationsManager` by invoking `registerNotificationsObjects()`.

```
NotificationsManager.registerConsequence(ConsequenceImpl.ID, new
    ConsequenceImpl());
```

To register the consequence when the handheld starts, perform this registration in a library project. See " Create a library project" on page 106 for more information.

Code example

Example: ConsequenceDemo.java

```
/**
 * ConsequenceDemo.java
 * Copyright (C) 2001-2004 Research In Motion Limited. All rights reserved.
*/

package com.rim.samples.docs.notifications;

import net.rim.device.api.synchronization.*;
import net.rim.device.api.notification.*;
import net.rim.device.api.system.*;
import net.rim.device.api.util.*;
import java.io.*;

public class ConsequenceDemo implements Consequence, SyncConverter {

    public static final long ID = 0xbd2350c0dfda2a51L;
    private static final int TYPE = 'n' << 24 | 'o' << 16 | 't' << 8 | 'd';
    private static final byte[] DATA = new byte[] {
            'm', 'y', '-', 'c', 'o', 'n', 'f', 'i',
```

```
        'g', '-', 'o', 'b', 'j', 'e', 'c', 't' };

private static final Configuration CONFIG = new Configuration(DATA);

private static final short BFlat = 466; // The actual value is 466.16.
private static final short TEMPO = 125;
private static final short d16 = 1 * TEMPO;
private static final short pause = 10; // 10 millisecond pause.
private static final short[] TUNE = new short[] {BFlat, d16, pause, BFlat};
private static final int VOLUME = 80; // Percentage volume.

public void startNotification(long consequenceID, long sourceID, long eventID,
    Object configuration, Object context) {
    LED.setConfiguration(500, 250, LED.BRIGHTNESS_50);
    LED.setState(LED.STATE_BLINKING);

    Alert.startAudio(TUNE, VOLUME);
    Alert.startBuzzer(TUNE, VOLUME);
}

public void stopNotification(long consequenceID, long sourceID, long eventID,
    Object configuration, Object context) {
    LED.setState(LED.STATE_OFF);
    Alert.stopAudio();
    Alert.stopBuzzer();
}

public Object newConfiguration(long consequenceID, long sourceID,
    byte profileIndex, int level, Object context) {
    return CONFIG;
}

public SyncObject convert(DataBuffer data, int version, int UID) {
    try {
        int type = data.readInt();
        int length = data.readCompressedInt();
        if ( type == TYPE ) {
            byte[] rawdata = new byte[length];
            data.readFully(rawdata);
            return new Configuration(rawdata);
        }
    } catch (EOFException e) {
        System.err.println(e);
    }
    return null;
}
public boolean convert(SyncObject object, DataBuffer buffer, int version) {
    boolean retval = false;
    if ( object instanceof Configuration ) {
        Configuration c = (Configuration)object;
        buffer.writeInt(TYPE);
        buffer.writeCompressedInt(c._data.length);
        buffer.write(c._data);
        retval = true;
    }
```

```
        return retval;
    }

    /* Inner class to store configuration profile. */
    private static final class Configuration implements SyncObject, Persistable {

        public byte[] _data;

        public Configuration(byte[] data) {
            _data = data;
        }
        public int getUID() {
            return 0;
        }
    }
}
```

12

Managing applications

- Application manager
- Managing code modules
- Managing code modules

Application manager

The virtual machine (VM) on BlackBerry Wireless Handhelds has an application manager that functions as the central dispatcher of operating system events for other Java applications.

The `net.rim.device.api.system.ApplicationManager` class enables applications to interact with the application manager to perform the following actions:

- interact with processes, such as retrieving the IDs for foreground applications
- post global events to the system
- lock or unlock the handheld, or determine whether the handheld is locked
- run an application immediately or at a specific time

To use any of the `ApplicationManager` methods, you must first retrieve a reference to the current application manager by invoking `getApplicationManager()`.

```
ApplicationManager manager = ApplicationManager.getApplicationManager();
```

Retrieve information about applications

Retrieve information about running processes by invoking `getVisibleApplications()` on the `ApplicationManager` class. For example you could write an system administration application that audits the state of the handheld to determine how long users spend using each application.

To retrieve an array of `ApplicationDescriptor` objects for visible applications that are running, invoke `getVisibleApplications()`. An `ApplicationDescriptor` object contains descriptive information for the application, such as its name, icon, position on the Home screen, and resource bundle. Use `ApplicationDescriptor` methods to retrieve this information. For example, to retrieve the name of a running application, invoke `getName()` on an application descriptor.

```
ApplicationManager manager = ApplicationManager.getApplicationManager();
ApplicationDescriptor descriptors[] = manager.getVisibleApplications();
// retrieve the name of a running application
String appname1 = descriptors[0].getName();
```

To retrieve an `ApplicationDescriptor` for the current application, invoke `ApplicationDescriptor.currentApplicationDescriptor()`.

```
ApplicationDescriptor descriptor =
ApplicationDescriptor.currentApplicationDescriptor();
String appname = descriptor.getName();
```

Post a global event

Use `ApplicationManager.postGlobalEvent()` as a basic mechanism to communicate with other processes.

Tip: You can also send and receive messages between processes using the runtime store. See " Sharing runtime objects between applications" on page 125 for more information.

To post a global event to a particular application, invoke `postGlobalEvent(int processId, long guid, int data0, int data1, Object object0, Object object1)`.

The `processID` parameter specifies the ID of the process to which to post the event. To retrieve a process ID, invoke `getProcessId(ApplicationDescriptor)`. The `guid` parameter specifies a global unique identifier (GUID) for the event. The `data` and `object` parameters specify additional information for the event.

To post a global event to all applications, use one of the following forms of `postGlobalEvent()`:

Method signature	Description
`boolean postGlobalEvent(long guid)`	posts a global event with a unique identifier
`boolean postGlobalEvent(long guid, int data0, int data1)`	posts a global event with additional data
`abstract boolean postGlobalEvent(long guid, int data0, int data1, Object object0, Object object1)`	posts a global event with additional integer and object data

Receive a global event

To receive a global event, implement the `net.rim.device.api.system.GlobalEventListener` interface and `GlobalEventListener.eventOccurred()`, which is invoked when a global event occurs. In its implementation of `eventOccurred()`, the application specifies which actions to perform when a global event is received.

Register the `GlobalEventListener` by invoking `Application.addGlobalEventListener(GlobalEventListener)`.

Lock the handheld

To determine whether a handheld is locked, invoke `ApplicationManager.isSystemLocked()`.

To lock a handheld, invoke `ApplicationManager.lockSystem(true)`.

If the user has set a password, the lock screen appears and the user must type this password to use the handheld again. If a password is not set, the keyboard lock screen appears and the user must double-click the trackwheel to use the handheld again.

To unlock a handheld, invoke `ApplicationManager.unlockSystem(true)`.

Run an application with different arguments

Create a new application descriptor

Use the existing `ApplicationDescriptor` as a template. Specify the arguments to use in `main()`.

```
ApplicationDescriptor template =
   ApplicationDescriptor.currentApplicationDescriptor();
String[] args = { "admin", "secure" };
ApplicationDescriptor newdescriptor = new ApplicationDescriptor(template, args);
```

The `ApplicationDescriptor` constructor has two other forms:

Method signature	Description
`ApplicationDescriptor(ApplicationDescriptor original, String name, String[] args)`	This form of the constructor enables you to specify a name for the new `ApplicationDescriptor`.
`ApplicationDescriptor(ApplicationDescriptor original, String name, String[] args, Bitmap icon, int position, String nameResourceBundle, int nameResourceId)`	This form of the constructor enables you to specify a name, and initial settings, including an application icon, a Home screen position, and the resource bundle and ID to use for the application title.

Run the application

Run the application using a new `ApplicationDescriptor` object.

```
ApplicationManager appmanager = ApplicationManager.getApplicationManager();
try {
   appmanager.runApplication(newdescriptor);
} catch(ApplicationManagerException) {
   //handle error
}
```

The `runApplication()` method creates a new process and invokes the exported `main()` method in the specified descriptor, using its arguments. The new process moves to the foreground if possible.

Run an application at a specified time

Invoke `scheduleApplication()` instead of `runApplication()`.

```
try {
   appmanager.scheduleApplication(newdescriptor, 1728000, false);
} catch(ApplicationManagerException) {
   //handle error
}
```

The `scheduleApplication()` method requires the following parameters:

- `an ApplicationDescriptor` object
- time at which to start the application, in milliseconds
- a Boolean value, where `true` indicates that the time is absolute (calculated from midnight, January 1, 1970 UTC) and `false` indicates that the time is relative to midnight local time

Note: The application does not run if the handheld is restarted or turned off before the specified time.

Managing code modules

The `CodeModuleManager` class, in the `net.rim.device.api.system` package, enables you to retrieve information about and manage code modules on the handheld.

A code module is a .cod file, the compiled archive of a single project in the IDE. To view a list of third-party applications installed on handhelds, in the handheld options, click **Applications**. Click the **Properties** menu item to view information about each application.

Retrieve module information

The `CodeModuleManager` class provides methods that enable applications to retrieve information about code modules on the handheld, such as the name, type, description, version, and creation date.

To retrieve a handle for the module, invoke `getModuleHandle()`. Provide as a parameter the name of the code module.

```
int handle = CodeModuleManager.getModuleHandle("test_module");
```

Invoke methods of the `CodeModuleManager` class to retrieve specific information. Provide the module handle as a parameter to these methods.

```
String name = CodeModuleManager.getModuleName( handle );
String vendor = CodeModuleManager.getModuleVendor( handle );
String description = CodeModuleManager.getModuleDescription( handle );
int version = CodeModuleManager.getModuleVersion( handle );
int size = CodeModuleManager.getModuleCodeSize( handle );
int timestamp = CodeModuleManager.getModuleTimestamp( handle );
```

Retrieve an array of handles

To retrieve an array of handles for all existing modules on the handheld, invoke `getModuleHandles()`.

```
int handles[] = CodeModuleManager.getModuleHandles();
String name = CodeModuleManager.getModuleName( handles[0]);
```

The `net.rim.device.api.system.CodeModuleManager` class provides methods for creating, saving, and deleting code modules. These capabilities enable an application on the handheld to receive .cod files wirelessly.

Code module manager methods

Method	Description
`int handle = CodeModuleManager.getModuleHandleForObject( anObject );`	This method retrieves the handle of the module in which an object class is defined.
`boolean library = CodeModuleManager.isLibrary( handle );`	This method determines whether a module is a library. This method returns `true` if the module is a library or `false` if the module is an application.
`int size = CodeModuleManager.getModuleHandleForObject( anObject);`	This method determines the size, in bytes, of the code that a module contains.
`ApplicationDescriptor descriptors[] = CodeModuleManager.getApplicationDescriptiors( handle );`	This method retrieves an array of all descriptors that a code module contains.

Create a module

Invoke `createNewModule()`. Provide the size of the module in bytes as a parameter.

```
int handle = CodeModuleManager.createNewModule( 3000 );
```

This method returns the module handle (or 0 if the module cannot be created).

To add data to the module when you create it, invoke the following form of `createNewModule()`. Provide as parameters the length in bytes of the entire module, a byte array to add to the module, and the `length` parameter specifies the number of bytes from the byte array to add to the start of the module.

```
static int createNewModule(int totalLength, byte[] data, int length);
```

Write data into a module

Invoke `writeNewModule()`. Provide a byte array of data as a parameter to this method.

```
Boolean success = CodeModuleManager.writeNewModule( handle, data, 0, data.length );
```

Tip: A module must have the correct format for a .cod file. You can write data into a code module in increments, as long as you know the offset at which to add data.

Save a module to the handheld database

To save a module to the handheld database, invoke `saveNewModule()`.

```
int result = CodeModuleManager.saveNewModule( handle );
```

The `saveNewModule()` method returns one of the result codes that are defined in the `CodeModuleManager` class, such as `CMM_OK` if the module is saved successfully.

Delete a module from the handheld database

Invoke `deleteModule()`. Provide as parameters the handle of the module to delete and a Boolean value to specify whether to delete the module and any data it contains or to delete the module only if does not have any associated data. If the module is in use, it is deleted the next time that the handheld is restarted.

```
int handle = CodeModuleManager.getModuleHandle("test_module");
if( handle != 0 ) {
   Boolean success = CodeModuleManager.deleteModule( handle, true );
}
```

13

Sharing runtime objects between applications

- Sharing runtime objects

Sharing runtime objects

BlackBerry handhelds use a runtime store to provide a central location in which applications can share runtime objects. By default, only applications that have been digitally signed by RIM can access data in the runtime store. Contact RIM for information on how to control access to your data.

Retrieve the runtime store

Invoke `RuntimeStore.getRuntimeStore()`.

```
RuntimeStore store = RuntimeStore.getRuntimeStore();
```

To add or retrieve runtime objects, invoke methods on `RuntimeStore`.

Note: The runtime store is not persistent. If the handheld is restarted, data in the runtime store is lost.

Add a runtime object

Invoke `RuntimeStore.put()`. Provide as parameters a unique long ID and the object to store.

```
RuntimeStore store = RuntimeStore.getRuntimeStore();
// create an object and a unique number to identify the object
String msg = "Some shared text";
long ID = 0x60ac754bc0867248L;
// put() throws an IllegalArgumentException if an object with the same ID exists
try {
   store.put( ID, msg );
} catch(IllegalArgumentException e) {
   // handle exception - an object with the sam
}
```

Replace a runtime object

Invoke `replace()`.

```
RuntimeStore store = RuntimeStore.getRuntimeStore();
String newmsg = "Some new text";
try {
   // returns the existing object with the specified ID if it exists, or null
   // otherwise
```

```
   Object obj = store.replace( 0x60ac754bc0867248L, newmsg);
} catch(ControlledAccessException e) {
   // handle exception - insufficient permissions
}
```

Retrieve a registered runtime object

Invoke `RuntimeStore.get()`. Provide as a parameter the object ID.

```
RuntimeStore store = RuntimeStore.getRuntimeStore();
// get() throws a ControlledAccessException if your application does not have read
   access to the specified object.
try {
   // get() returns the objectm with the specified ID if it exists, or null
   // otherwise
   Object obj = store.get(0x60ac754bc0867248L);
} catch(ControlledAccessException e) {
   //handle exception
}
```

Retrieve an unregistered runtime object

Invoke `RuntimeStore.waitFor()` to wait for an object to be registered.

```
RuntimeStore store = RuntimeStore.getRuntimeStore();
try {
   Object obj = store.waitFor(0x60ac754bc0867248L);
} catch(ControlledAccessException e) {
   //handle exception - insufficient permissions
} catch(RuntimeException e) {
   //handle exception - time out
}
```

Note: If the object with the specified ID does not exist, `waitFor()` blocks for a maximum of `MAX_WAIT_MILLIS`. The `waitFor()` method throws a `RuntimeException` if the object is not registered by this time.

Glossary

A

ALX
: Application Loader XML

API
: application programming interface

APN
: Access Point Name

C

CA
: Certificate Authority

CDMA
: Code Division Multiple Access

CHAP
: Challenge Handshake Authentication Protocol

cHTML
: Compact Hypertext Markup Language

CLDC
: Connected Limited Device Configuration

CPU
: central processing unit

D

DES
: Data Encryption Standard

DNS
: Domain Name System

G

GIF
: Graphics Interchange Format

GPRS
: General Packet Radio Service

GUI
: graphical user interface

GUID
: globally unique identifier

H

HTML
: Hypertext Markup Language

HTTP
: Hypertext Transfer Protocol

HTTPS
: Hypertext Transfer Protocol over Secure Socket Layer

I

i18n
: internationalization

IDE
: integrated development environment

iDEN
: Integrated Digital Enhanced Network

IMEI
: International Mobile Equipment Identity

IMSI
: International Mobile Subscriber Identity

I/O
: input/output

IP
: Internet Protocol

IPPP
: IP Proxy Protocol

ISDN
: Integrated Services Digital Network

J

J2ME
: Java 2 Platform, Micro Edition

J2SE
: Java 2 Platform, Standard Edition

JAD
: Java Application Descriptor

JAR
: Java Archive

JDE
: Java Development Environment

JPEG
: Joint Photographic Experts Group

JRE
: Java Runtime Environment

K

KB
: kilobytes

KVM
: Kilobyte virtual machine

L

LAN
: local area network

LDAP
: Lightweight Directory Access Protocol

LTPA
: Lightweight Third-Party Authentication

M

MB
: megabyte

MHz
: megahertz

MIDlet
: MIDP application

MIDP
: Mobile Information Device Profile

MIME
: Multipurpose Internet Mail Extensions

MSISDN
: Mobile Station ISDN

O

OCSP
: Online Certificate Status Protocol

P

PAP
: Password Authentication Protocol

PDA
: personal digital assistant

PIM
: personal information management

PIN
: personal identification number

PNG
: Portable Network Graphics

R

RAM
: random access memory

RRC
: Radio Resource Control

RTC
: real-time clock

S

SDK
: software development kit

SIM
: Subscriber Identity Module

SMS
: Short Message Service

SRAM
: static random access memory

SRP
: Service Relay Protocol

SSL
: Secure Sockets Layer

T

TCP
: Transmission Control Protocol

TCP/IP
: Transmission Control Protocol/Internet Protocol

TIFF
: Tag Image File Format

TLS
: Transport Layer Security

U

UDP
: User Datagram Protocol

UI
: user interface

URI
: Uniform Resource Identifier

URL
: Uniform Resource Locator

UTC
: Universal Time Coordinate

V

VM
: virtual machine

W

WAP
: Wireless Application Protocol

WBMP
: wireless bitmap

WML
: Wireless Markup Language

WMLC
: Wireless Markup Language Compiled

WTLS
: Wireless Transport Layer Security

X

XHTML
: Extensible Hypertext Markup Language

XML
: Extensible Markup Language

Index

A

addCall(), PhoneLogs class, 71
addElement(), Vector class, 87
addGlobalEventListener(), Application class, 120
addMenuItem(), ApplicationMenuItemRepository class, 77
addPhoneListener(), Phone class, 70
addRecordStore(), RecordStore class, 79
address book
  about, 25
  converting to serial formats, 28
  creating contacts, 25
  importing contacts, 28
  invoking, 75
  opening ContactList, 25
  removing contacts, 29
  retrieving contact information, 27
  saving contacts, 27
administrative control
  about, 81
APIs
  invocation, 75
  messaging, 11
  notifications, 105
  persistence, 79
  PIM, 23
  service book, 103
  synchronization, 93
Application class, addGlobalEventListener(), 120
application descriptor, different arguments, 120
application manager, virtual machine, 119
ApplicationDescriptor class
  about, 120
  currentApplicationDescriptor(), 120
ApplicationManager class
  about, 119
  getVisibleApplications(), 119
  isSystemLocked(), 120
  lockSystem(), 120
  postGlobalEvent(), 120
  runApplication(), 121
  scheduleApplication(), 121
  unlockSystem(), 120
ApplicationMenuItem class
  about, 76
  extending, 76
  overriding constructor, 76
  toString(), 76
ApplicationMenuItemRepository class
  about, 76
  addMenuItem(), 77
  getInstance(), 77
applications
  auto-run on startup, 95
  different arguments, 120
  event source, 106
  inter-process communication, 120
  managing, 119
  retrieving information, 119
  scheduling, 121
  system modules, 95
  See also code modules
appointments, *See* calendar
attachments
  about, 19
  registering a handler, 20
  retrieving contents, 20
  retrieving information, 20
  sending, 21
audio formats, supported, 113
auto-run, applications, 95

B

backup
  about, 81
  implementing, 93
  supporting, 94
Backup and Restore
  about, 93
BlackBerry applications
  adding menu items, 76
  starting, 75
Boolean data type, persistence, 81
browser
  about, 47
  API overview, 47
Browser class
  getDefaultSession(), 47
  getSession(), 47
browser content
  filtered, 64
  retrieving, 51

browser fields, about, 48
browser sessions, retrieving, 47
BrowserContentProvider class
  about, 58
  getAccept(), 59
  getSupportedMIMETypes(), 59
BrowserContentProvider interface
  getBrowserContent(), 59
BrowserField class, finishLoading(), 51
BrowserPageContext interface, about, 59
BrowserSession class, displayPage(), 48
Byte data type, persistence, 81

C

calendar
  adding appointment information, 37
  converting to serial formats, 39
  creating a recurring appointment, 38
  creating an appointment, 37
  invoking, 75
  opening a list, 37
  retrieving appointment information, 39
  saving an appointment, 38
  See also PIM
CalendarArguments class, about, 75
call logs
  adding, 71
  code example, 72
  replacing, 71
callAt(), PhoneLogs class, 71
cancelAllDeferredEvents(), NotificationsManager class, 108
cancelDeferredEvent(), NotificationsManager class, 108
cancelImmediateEvent(), NotificationsManager class, 107
Character data type, persistence, 81
.cod files, compiled projects, 9
code examples
  BasicMail.java, 17
  BrowserFieldSampleApp.java, 52
  BrowserPlugin.java, 60
  ConsequenceDemo.java, 115
  ContactsDemo.java, 29
  DemoAppMenuItem.java, 77
  DemoOptionsProvider.java, 44
  EventDemo.java, 40
  NotificationsDemo.java, 108
  PhoneLogsDemo.java, 72
  Protocol.java, 64
  Restaurants.java, 87
  RestaurantsSync.java, 95
  TaskDemo.java, 35
  UserInfo.java, 83
code modules
  creating, 121
  deleting, 123
  methods, 122
  retrieving handle arrays, 122
  retrieving handles, 122
  retrieving information, 121
  saving, 123
  writing, 123
code signing
  about, 5
  registering, 8
  requesting signatures, 9
  verification, 6
CodeModuleManager class
  about, 121
  createNewModule(), 122
  deleteModule(), 123
  getApplicationDescriptor(), 122
  getModuleHandle(), 122
  getModuleHandleForObject(), 122
  getModuleHandles(), 122
  isLibrary(), 122
  saveNewModule(), 123
  writeNewModule(), 123
commit(), PersistentObject class, 82
conference calls, about, 71
ConferencePhoneCallLog class
  constructor, 71
  getParticipantAt(), 71
Consequence class
  newConfiguration(), 114
  startNotification(), 113
  stopNotification(), 113
Consequence interface
  about, 112
ContactList class
  about, 25
  closing, 27
  opening, 25
convert(), SyncConverter class, 114
createNewModule(), CodeModuleManager class, 122
.csl files, required signatures, 9
.cso files, optional signatures, 9
currentApplicationDescriptor(), ApplicationDescriptor class, 120
custom consequences, code example, 115
custom notifications
  about, 110
  defining, 111
  user profile settings, 114

custom objects, managing, 85

D

data
  persistent storage, 82
data integrity, 81
data types
  persistence, 81
databases, creating persistent, 82
deferred events
  cancelling, 108
  cancelling all, 108
  code example, 108
  triggering, 107
deferredEventWasSuperseded(),
  NotificationsEngineListener interface, 111
deleteCall(), PhoneLogs class, 72
deleteModule(), CodeModuleManager class, 123
deleting, persistent databases, 83
display characteristics, specifying, 59
displayBrowserField(), RenderingApplication class, 48
displayPage(), BrowserSession class, 48
DTMF tones, queueing, 70

E

email, *See* messaging
enableSynchronization(), SerialSyncManager class, 94
entry points, alternate, 95
enumerateRecords(), RecordStore class, 80
eventOcurred(), GlobalEvent interface, 120
events
  adding, 105
  calendar, 37
  cancelling, 107
  global, 120
  messaging, 13
  phone, 70
  service book, 104
  triggering deferred, 107
examples
  BasicMail.java, 17
  BrowserFieldSampleApp.java, 52
  BrowserPlugin.java, 60
  ConsequenceDemo.java, 115
  ContactsDemo.java, 29
  DemoAppMenuItem.java, 77
  DemoOptionsProvider.java, 44
  EventDemo.java, 40
  NotificationsDemo.java, 108
  PhoneLogsDemo.java, 72
  Protocol.java, 64
  Restaurants.java, 87
  RestaurantsSync.java, 95
  TaskDemo.java, 35
  UserInfo.java, 83

F

file extensions
  .cod files, 9
  .csl files, 9
  .cso files, 9
filtering URLs, 64
finishLoading(), BrowserField class, 51
FolderEvent class, about, 13
folders
  listing, 18
  managing, 18
  saving messages, 19
  searching, 19

G

get(), RuntimeStore class, 126
getAccept(), BrowserContentProvider class, 59
getActiveCall(), Phone class, 69
getApplicationDescriptor(), CodeModuleManager class,
  122
getBrowserContent(), BrowserContentProvider
  interface, 59
getBrowserField(), RenderingApplication interface, 51
getContents(), PersistentObject class, 82
getDefaultSession(), Browser class, 47
getDisplayPhoneNumber(), PhoneCall class, 69
getElapsedTime(), PhoneCall class, 69
getInstance()
  ApplicationMenuItem class, 77
  OptionsProvider interface, 43
  SerialSyncManager class, 94
getModuleHandle(), CodeModuleManager class, 122
getModuleHandles(), CodeModuleManager class, 122
getModuleManagerForObject(), CodeModuleManager
  class, 122
getParticipantAt(), ConferencePhoneCallLog class, 71
getParticipants(), PhoneCallLog class, 71
getRecord(), RecordStore class, 79
getRenderingOptions(), RenderingApplication
  interface, 49
getRuntimeStore(), RuntimeStore class, 125
getSession(), Browser class, 47
getStatus(), PhoneCall class, 69
getSupportedContentTypes(), Manager class, 113
getSupportedMIMETypes(), BrowserContentProvider
  class, 59
getVisibleApplications(), ApplicationManager class, 119
global events, posting, 120
GlobalEventListener interface, about, 120

H

handheld options
about, 43
adding, 43
code example, 44
registering, 43
storing data persistently, 44
using public methods, 44
handhelds
locking, 120
unlocking, 120
Hashtable data type, persistence, 81
HTTP filters, about, 64
HTTPFilterRegistry class, registerFilter(), 64

I

immediate events
cancelling, 107
code example, 108
triggering, 107
initialization projects, creating, 95
Integer data type, persistence, 81
integrity, data, 81
Intellisync, about, 93
intercepting, URLs, 64
inter-process communication, global events, 120
Invoke class, invokeApplication(), 75
invokeApplication(), Invoke class, 75
isLibrary(), CodeModuleManager class, 122
isOutgoing(), PhoneCall class, 69
isSystemLocked(), ApplicationManager class, 120

L

lastElement(), Vector class, 87
library projects, creating, 106
locking, handhelds, 120
lockSystem(), ApplicationManager class, 120
Long data type, persistence, 81

M

Manager class, getSupportedContentTypes(), 113
managing, applications, 119
memo pad, invoking, 75
MemoArguments class, about, 75
menu items
adding, 76
adding to BlackBerry applications, 76
registering, 77
specifying behavior, 76
specifying position, 76
specifying text, 76
MessageArguments class, about, 75
MessageEvent class, about, 13
messages
about, 11
attachments, 19
creating, 11
managing events, 13
managing folders, 18
multipart, 12
reading, 14
receiving, 13
replying, 16
storing, 12
MIME types
listing accepted, 59
supporting additional, 58
Mobile Media API, about, 113

N

negotiateDeferredEvent(), Notifications manager class, 107
newConfiguration(), Consequence class, 114
notifications
about, 105
adding events, 105
canceling events, 107
custom, 110
custom system, 112
custom system notifications, 112
deferred events, 107
defining custom, 111
holstering, 111
immediate events, 107
listener, 111
registering listener, 112
superseded, 111
triggering deferred events, 107
tunes, 113
NotificationsEngineListener interface, 111
about, 110
deferredEventWasSuperseded(), 111
notificationsEngineStateChanged(), 111
registerNotificationsEngineListener(), 112
notificationsEngineStateChanged(), NotificationsEngineListener interface, 111
NotificationsManager class
cancelDeferredEvent(), 108
cancelImmediateEvent(), 107
code example, 108
negotiateDeferredEvent(), 107
registerNotificationsObject(), 115
registerSource(), 106

triggerImmediateEvent(), 107
numberOfCalls(), PhoneLogs class, 71

O

Object data type, persistence, 81
openRecordStore(), RecordStore class, 79
options
See handheld options
OptionsProvider interface
about, 43
getInstance(), 43
registerOptionsProvider(), 43

P

persistence
BlackBerry APIs, 80
code examples, 87
creating databases, 82
custom objects, 85
limited memory, 80
MIDP API, 79
retrieving data, 82
retrieving objects, 87
saving data, 82
saving objects, 86
persistent data, managing, 81
persistent databases, deleting, 83
PersistentObject class
about, 82
commit(), 82
getContents(), 82
setContents(), 82
PersistentStore class, about, 80
phone calls
about, 69
adding DTMF tones, 70
duration, 69
retrieving, 69
status, 69
Phone class
addPhoneListener(), 70
getActiveCall(), 69
removePhoneListener(), 70
phone listener, registering, 70
phone logs
about, 70
missed calls folder, 71
normal calls folder, 71
retrieving, 71
phone, invoking, 75
PhoneArguments class, about, 75
PhoneCall class
about, 69
getDisplayPhoneNumber(), 69
getElapsedTime(), 69
getStatus(), 69
isOutgoing(), 69
PhoneCallLog class
constructor, 71
getParticipants(), 71
PhoneCallLogID class
about, 71
PhoneListener interface, about, 70
PhoneLogs class
addCall(), 71
callAt(), 71
deleteCall(), 72
getInstance(), 71
swapCall(), 71
PIM
fields, 23
items, 23
lists, 23
See also Address Book, Calendar, Tasks
postGlobalEvent(), ApplicationManager class, 120
proceedWithDeferredEvent(),
NotificationsEngineListener interface, 111
put(), RuntimeStore class, 125

R

receiving, email messages, 13
record stores
about, 79
adding, 79
opening, 79
retrieving, 79
retrieving all, 80
RecordStore class
addRecordStore(), 79
enumerateRecords(), 80
getRecord(), 79
openRecordStore(), 79
registerFilter(), HTTPFilterRegistry class, 64
registering
attachment handler, 20
notification event source, 105
options items, 43
startup, 106
synchronization collections, 94
registerNotificationsEngineListener(),
NotificationsEngineListener class, 112
registerNotificationsObjects(), NotificationsManager
class, 115
registerOptionsProvider(), OptionsProvider interface, 43

registerSource(), NotificationsManager class, 106
removePhoneListener(), Phone class, 70
rendering
    separate threads, 48
rendering options
    specifying, 49
rendering providers
    registering, 58
RenderingApplication class, displayBrowserField(), 48
RenderingApplication interface,
    getRenderingOptions(), 49
replace(), RuntimeStore class, 125
restore
    about, 81
    impementing, 93
    supporting, 94
retrieving
    attachment contents, 20
    attachment information, 20
root database, persistence, 82
runApplication(), ApplicationDescriptor class, 121
runtime APIs, about, 5
runtime objects
    retrieving, 125
    sharing, 125
runtime storage, 5
runtime store
    replacing objects, 125
    retrieving objects, 126
    unregistered objects, 126
RuntimeStore class
    get(), 126
    getRuntimeStore(), 125
    put(), 125
    replace(), 125
    waitFor(), 126

S

saveNewModule(), CodeModuleManager class, 123
scheduleApplication(), ApplicationDescriptor class, 121
scheduling, applications, 121
security
    See also code signing
security, about, 81
security, handheld lock, 120
sending
    attachments, 21
    messages, 15
serial formats
    appointments, 39
    contacts, 28
    tasks, 34
SerialSyncManager class
    enableSynchronization(), 94
    getInstance(), 94
service books
    about, 103
    events, 104
service records
    about, 103
setContents(), PersistentObject class, 82
Short data type, persistence, 81
signing, *See* code signing
startNotification(), Consequence class, 113
stopNotification(), Consequence class, 113
StoreEvent class, about, 13
String data type, persistence, 81
swapCall(), PhoneLogs class, 71
SyncCollection class
    registering, 94
SyncConverter class
    convert(), 114
    notifications, 112
synchronization
    about, 93
    collections, 94
    initializing, 95
    objects, 94
    registering collections, 94
SyncObject interface, about, 94

T

TaskArguments class, about, 75
tasks
    adding information, 32
    code example, 35
    converting to serial formats, 34
    creating, 32
    invoking, 75
    removing, 34
    retrieving information, 33
to do, *See* tasks
toString(), ApplicationMenuItem class, 76
triggerImmediateEvent(), NotificationsManager class,
    107
tunes
    audio formats, 113
    creating, 113

U

unique keys, creating, 82
unlocking, handhelds, 120
unlockSystem(), ApplicationManager class, 120
user profile settings, custom notifications, 114

V

Vector class
    addElement(), 87
    lastElement(), 87
Vector data type, persistence, 81

W

waitFor(), RuntimeStore class, 126
web content
    browser fields, 48
    displaying, 47
web pages
    displaying, 47
    requesting, 48
writeNewModule(), CodeModuleManager class, 123